2009

Algorithms for testing equivalence of finite automata, with a grading tool for JFLAP

Daphne Norton

Rochester Institute of Technology - Department of Computer Science

M.S. Project Report:

Algorithms for Testing Equivalence of Finite Automata, with a Grading Tool for JFLAP

Daphne A. Norton
http://www.cs.rit.edu/~dan0337
dan0337@cs.rit.edu

March 22, 2009

Chair: Prof. Edith Hemaspaandra	Date
Reader: Prof. Ivona Bezakova	Date
Observer: Prof. Christopher M. Homan	Date

Contents

List of Figures

1 Abstract

A wide variety of algorithms can be used to determine the equivalence of two Deterministic Finite Automata (DFAs) and/or Nondeterministic Finite Automata (NFAs). This project focuses on three key areas:

1. A detailed discussion of several algorithms that can be used to prove the equivalence of two DFAs (and/or NFAs, since every NFA has an equivalent DFA), with an analysis of the time complexity involved in each case.

2. Modifications to a few of these algorithms to produce a 'witness' string if the two automata are not equivalent. This string is accepted by one of the automata, but not by the other, so it serves as a clear demonstration of why the two automata are inequivalent.

3. A Java implementation of a couple of efficient algorithms to prove equivalence. The code is designed specifically to work with JFLAP, the Java Formal Language and Automata Package. JFLAP is a popular program from Duke University which can be used to demonstrate and manipulate models such as finite automata. JFLAP software allows students to enter finite automata via an easy-to-use GUI, and this project incorporates functionality so that instructors can grade homework assignments and/or allow students to receive detailed feedback in the form of a witness.

2 Preliminaries

2.1 Definitions

This section presents some of the basic concepts and definitions used in the body of this paper. Programmers are generally quite familiar with the regular expressions which DFAs and NFAs represent, since they frequently use regular expressions to parse and manipulate strings. However, they may not have encountered these expressions represented as state/transition models.

A Deterministic Finite Automaton (DFA) is formally defined as a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where:

1. Q is a finite set of states.

2. Σ is a finite alphabet (the set of symbols used in the input strings).

3. δ is a transition function which maps a state and an input symbol to the next state. δ: $Q \times \Sigma \rightarrow Q$.

4. q_0 is the start state, where $q_0 \in Q$.

5. F is the set of "accepting" (final) states, where $F \subseteq Q$.

A Nondeterministic Finite Automaton (NFA) is similar, but more flexible. It also consists of a 5-tuple $(Q, \Sigma, \delta, q_0, F)$. The only difference is that the transition function is defined as δ: $Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow P(Q)$, where ε is the empty string, and $P(Q)$ is the power set of Q. While a DFA always moves to a single state on a single input symbol, the NFA is nondeterministic, so it may have a choice. Starting from a given state on a particular input symbol, the NFA could have transitions to zero, one, or any of multiple states. The concept behind the NFA is that it "nondeterministically" guesses an allowable possibility for a given input. An NFA may also transition on the empty string, if so specified.

The DFA (or NFA) reads the input string one character at a time, and moves to the next appropriate state according to the transition function (or guesses the next state, if there are multiple options). A DFA "accepts" an input string if the transition on the last symbol goes to an accepting state. An NFA accepts the input string if at least one way exists to process this string (including possible transitions on the empty string along the way) such that the end result is an accepting state. Two finite automata are considered "equivalent" if they have the same alphabet and accept the same set of strings (i.e., the same language). Note that the DFA is simply a special, more restricted, case of the NFA. To convert a DFA to an NFA, it is only necessary to change the type of the transition function output from a state to a *set* of states. Moreover, every NFA can be converted to an equivalent DFA (see the section on the subset construction below); both of these models accept the same class of languages, the regular languages.

$\delta(q, a)$ itself takes just one input symbol a, or ε, at a time. Therefore, for convenience, we define the extended transition function $\hat{\delta}$ to take an input string w and follow the appropriate series of transitions to a new set of possible states. It is assumed that $w \in \Sigma^*$, where Σ^* consists of a string of zero or more symbols from Σ. The formal definition for $\hat{\delta}(q, w)$ in DFAs is inductive:

Basis: $|w| = 0$, meaning $w = \varepsilon$. For all $q \in Q$, $\hat{\delta}(q, \varepsilon) = q$.

Induction: $|w| > 0$. Let $w = xa$ where $x \in \Sigma^*$ and $a \in \Sigma$. Then $\hat{\delta}(q, w) = \delta(\hat{\delta}(q, x), a)$.

A state in a DFA is considered equivalent to another state if no string exists which is accepted by following the transitions from one of these states, but rejected by following the transitions from the other state. In other words, the same result (accept or reject) is always obtained from both states for the same input string. On the other hand, if the states are *not* equivalent, they are "distinguishable" by some string. Formally, states p and q are equivalent if for every $w \in \Sigma^*$, $\hat{\delta}(p, w) \in F$ iff $\hat{\delta}(q, w) \in F$, where F is the set of final (accepting) states. Observe that for NFAs, the situation is more complex. Two states are equivalent if for every string w, the transitions from the first state can lead to an accepting state iff the transitions from the second state can result in an accepting state. To be equivalent, both states must lead to a possible scenario where w is accepted, or both states must have no path on w leading to any accepting state.

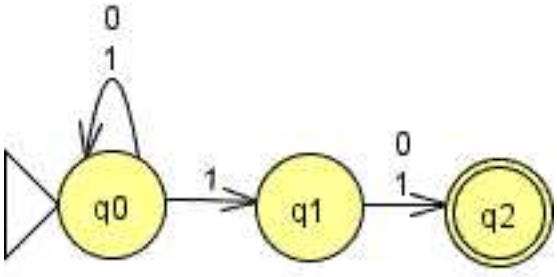

Figure 1: An NFA which accepts strings with a 1 in the second position from the end.

A "minimized" DFA contains the least possible number of states necessary to accept its language. The minimal DFA is unique for a given language, although the states may be arbitrarily labeled with different names. To minimize a DFA, each group of equivalent states must be combined into a single state, and all unreachable states (those not accessible on any input string from the start state) must be removed. (Formally, state q is reachable if an input string w exists such that $\hat{\delta}(q_0, w) = q$.) Note that an NFA may potentially have fewer states than the minimal DFA for the same language. The format of the diagrams will be discussed more in the next section, but one simple example is the NFA in Figure 1, which accepts all strings with a 1 in the second position from the end. It has three states, and it accepts strings such as 11, 010, and 000111. The minimal DFA for this language requires four states, as shown in Figure 2. Notice that it has one final state for strings ending in 0, and another final state for strings ending in 1. Sipser [29] gives a somewhat bigger example for strings with a 1 in the third position from the end (p. 51): the NFA only has four states, but the minimal DFA requires eight. As we will see, there can be an exponential difference in size between the NFA and the minimal DFA.

In addition to the DFA and NFA described above, a wide variety of automata exists in the literature. Models have been tailored to a number of specialized purposes. However, this paper will focus on the standard models, as DFAs and NFAs are the most common automata used for regular languages.

2.2 JFLAP Diagrams of Finite Automata

The most intuitive way for a student to understand a finite automaton is to look at a diagram. The examples in this paper were drawn in JFLAP. Figure 3 is a DFA which accepts the language consisting of the string 001. It contains five

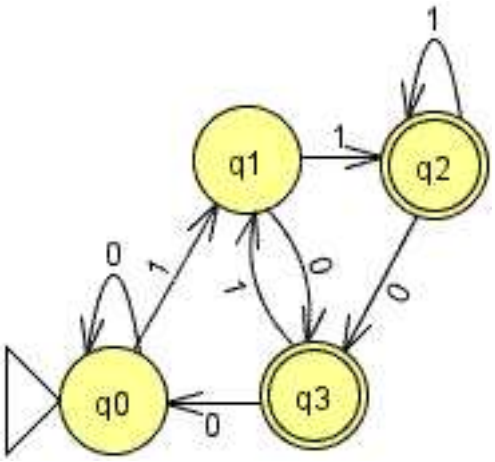

Figure 2: A DFA which accepts strings with a 1 in the second position from the end.

states, $\{q_0, q_1, q_2, q_3, q_4\}$. In JFLAP, notice that the start state (q_0) is indicated with a large triangle. Transitions are indicated with arrows, and the alphabet in this example includes the two symbols 0 and 1. Some of the arrows, such as the one from q_3 to q_4, represent transitions for both symbols 0 and 1; the symbols appear stacked on top of each other in the figure. The only accepting (final) state here is q_3, and it is distinguished by a double circle. A little experimentation on the part of the reader (by following the transitions from the start state on a given input) will show that 001 is the only string which is accepted. Any other string will either not reach q_3, and/or become stuck in the "dead" state, q_4. The JFLAP software does not require the dead state, since the transitions leading to this state could simply be left out of the automaton, and q_4 eliminated. Any input that has no path to follow would be rejected. However, the strict definition of the DFA normally specifies that the transition function is total, meaning that it is defined for all possible inputs. Therefore, transitions on each letter in the alphabet should be shown exiting each state in the DFA, and a dead state should be employed where necessary.

A couple of other simple examples we will refer to later: the DFA in Figure 4 accepts the language consisting of the string 00, and Figure 5 accepts 01. The only difference between these two diagrams, aside from the naming of the states, is that the transitions exiting q_b and q_f flip the 0 vs. the 1, so that the appropriate string reaches the accepting state.

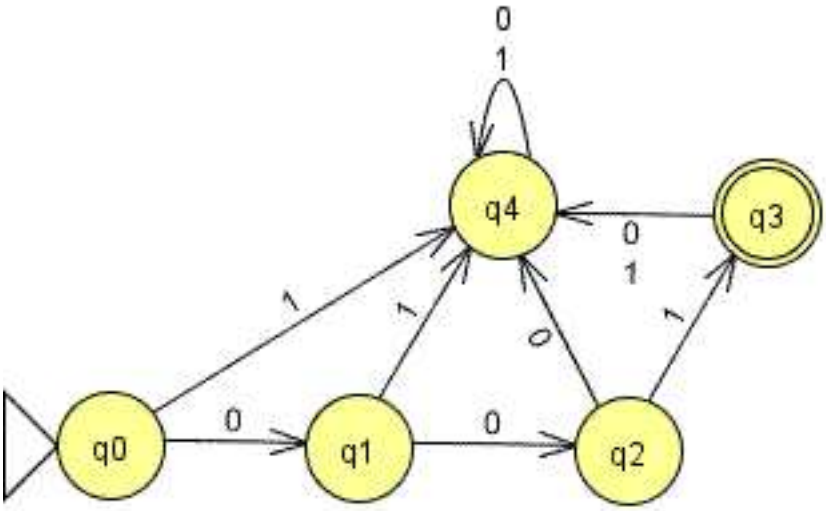

Figure 3: A DFA which accepts only the string 001.

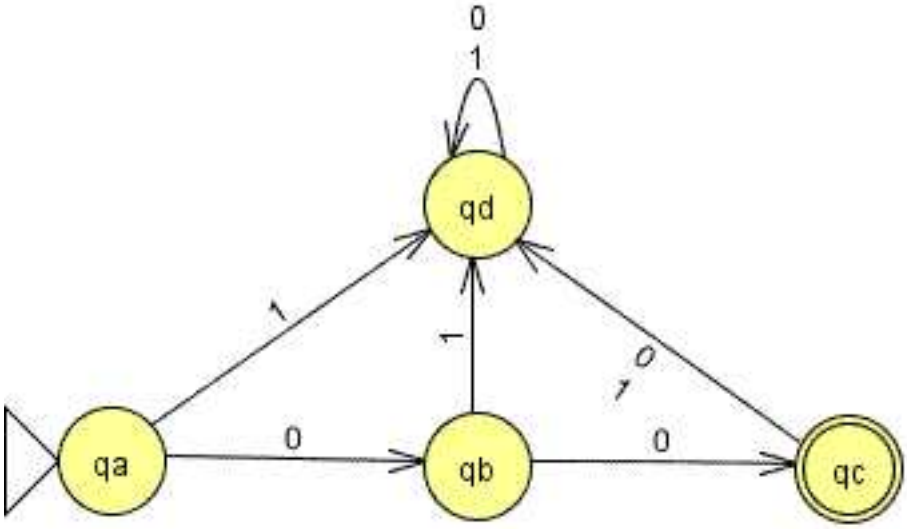

Figure 4: A DFA which accepts only the string 00.

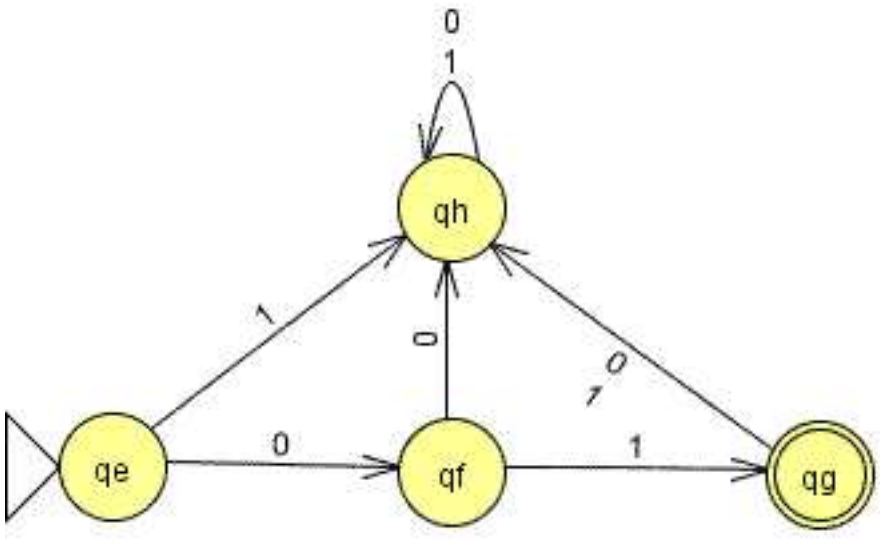

Figure 5: A DFA which accepts only the string 01.

3 Determining Equivalence of Finite Automata

3.1 Equivalence to Grade Assignments

The process of grading a student's assignment is essentially the process of determining if the DFA or NFA they submitted is equivalent to the correct answer. Although student examples of finite automata will tend to have a small n, where n is the number of states in the automaton, it is still desirable to use an efficient algorithm to grade the answers. The same code could then be used to handle larger cases. Moreover, even small cases may run slowly if part of the algorithm takes exponential time in n; for example, the subset construction may be a good thing to avoid if it is not needed. Several different algorithms will be considered below.

In an article about their Python tool called FAdo (for "Finite Automata devoted oracle"), Moreira and Reis stress the importance of being able to correct students' answers immediately to help them to learn. They use an unusual DFA "canonical form" in their program to test equivalence (via isomorphism) in what they consider linear time:

> "we can minimise the two automata and verify if the two minimised DFA's are *isomorphic* (i.e are the same up to renaming of states). For verify isomorphism we developed a canonical form for DFA's. Given a DFA we can obtain a unique string that represents it. Let

> Σ be ordered (p.e, lexicographically), the set of states is reordered in the following manner: the initial state is the first state; following Σ order, visit the states reachable from initial state in one transition, and if a state was not yet visited, give it the next number; repeat the last step for the second state, third state, ...until the number of the current state is the total number of states in the new order. For each state, a list of the states visited from it is made and a list of these lists is constructed. The list of final states is appended to that list. The result is a *canonical form*. If a DFA is minimal, the alphabet and its canonical form uniquely represent a regular language. For test of equivalence it is only needed to check whether the alphabets and the canonical forms are the same, thus having linear costing time." ([23], p. 336, sic)

However, note that the total time is not truly linear. The linear cost is actually in addition to the time it takes to first minimize the DFAs. NFAs would need to be converted to (minimized) DFAs for this to work. (Moreover, better algorithms are available for testing graph isomorphism of DFAs. For instance, the DFAEqualityChecker in JFLAP does not require conversion to a canonical form.) This is not the best choice of algorithm to use. Perhaps a more interesting approach is that when the student answers a question incorrectly, Moreira and Reis suggest and implement a way to provide detailed feedback to the student using what they call "witnesses," or strings which are in one language but not in the other:

> "Instead of a simple statement that an answer is wrong, we can exhibit a word that belongs to the language of the solution, but not to the language of the answer (or vice-versa). A *witness* of a DFA, can be obtained by finding a path from the initial state to some final state. If no *witness* is found, the DFA accepts the empty language. Given A and B two DFA's, if $\neg A \cap B$ or $A \cap \neg B$ have a witness then A and B are not equivalent. If both DFA's accept the empty language, A and B are equivalent." ([23], p. 336, sic)

It is very helpful to the student if they can see examples showing why their automaton needs to be corrected.

3.2 Converting NFAs to DFAs: The Subset Construction

It is generally necessary to convert an NFA to DFA format before checking equivalence of automata, since the algorithms discussed below only accept DFAs as input. The standard algorithm for conversion from an NFA to a DFA is called the "subset construction." The idea is to identify the set of possible states which the NFA can transition to on a single input, and to turn this set into a single state in the DFA. The resulting DFA can be much larger than the NFA; if Q is the set of states in the original NFA, the power set $P(Q)$ is of size $2^{\|Q\|}$, so the DFA may contain up to $2^{\|Q\|}$ states. Since this is an exponentially larger number

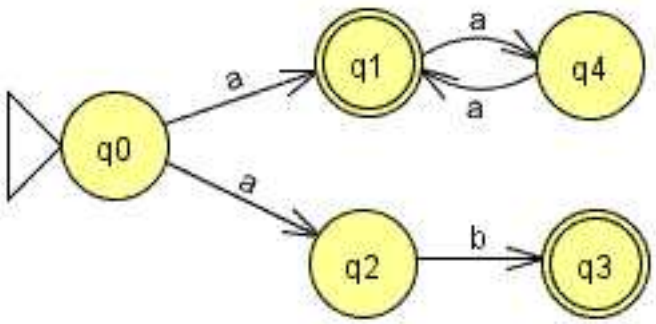

Figure 6: An NFA which accepts $ab + (a(aa)^*)$.

of states, we only construct those states which are actually accessible (reachable on some input from the start state). Hopcroft, Motwani, and Ullman argue that "the DFA in practice has about as many states as the NFA, although it often has more transitions [because a transition must exist for every input symbol for every state in the DFA]... We often can avoid the exponential-time step of constructing transition-table entries for every subset of states if we perform 'lazy evaluation' on the subsets" ([16], pp. 60-62), so the runtime may not necessarily be exponential unless we approach the worst case. (*Lazy evaluation* is a programming strategy where objects are not created and values are not calculated until they are actually required for use by the algorithm. This can potentially reduce both memory requirements and runtime.)

Here is a simple example of how the subset construction process works. Figure 6 shows an NFA which accepts the string ab as well as strings which consist of an odd number of a symbols, such as $\{a, aaa, aaaaa, \ldots\}$. This automaton is clearly nondeterministic because two different transitions exit state q_0 on the same symbol, a. The alphabet $\Sigma = \{a, b\}$. The DFA will use the same alphabet as the NFA. Set the DFA start state equal to the set $\{q_0\}$ – there is only one possible initial state of the NFA. Then, on input a, we can move to (guess) either of two different NFA states, so we construct the next DFA state using the set $\{q_1, q_2\}$ and add the a transition to it, coming from the start state. See Figure 7.

Although JFLAP does not require it, note that the strict definition for DFAs requires that a transition be defined for every input symbol of the alphabet on every state. Therefore, since transitions do not exist from a given set of NFA states (such as $\{q_0\}$ here) on some input symbol (b), we create a non-accepting dead state $\emptyset = \{\}$ in the corresponding DFA and transition to this state on b. Since the symbol $\emptyset$ is not available in JFLAP, the dead state is labelled *qphi* in the diagram. Note that the new JFLAP version 6.4 calls this a *trap state*. To make the trap state "dead," all transitions out of $\emptyset$ loop right back into $\emptyset$, so no string will be accepted once this state has been reached. (Observe that if we

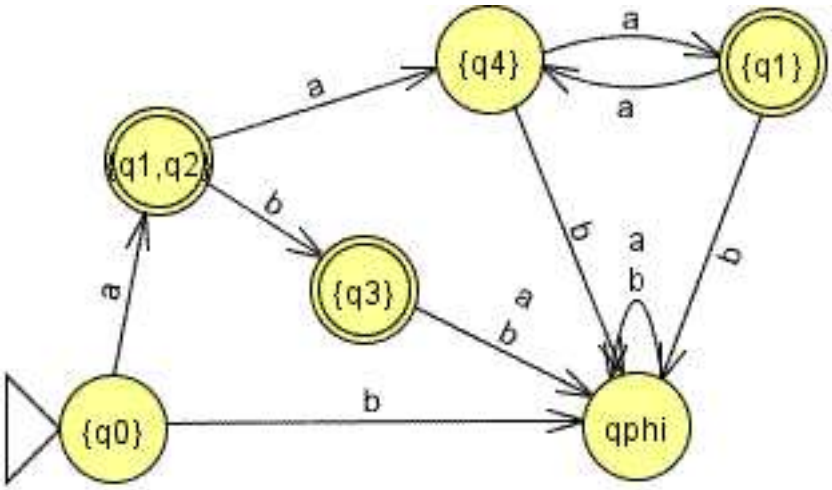

Figure 7: The DFA produced by the subset construction on the NFA in Figure 6.

were not taking a 'lazy evaluation' approach to create states, a state for the set $\emptyset$ would be created automatically since it is an element of the power set of the states of the NFA.)

At this point, we have processed all transitions exiting the start state. The next step is to track each state we have added to the DFA and continue to loop through them to process each possible transition. So far, the only new state (other than the dead state) is $\{q_1, q_2\}$. We try each letter of the alphabet to see where it leads in the NFA. Inputting a to the combination of states q_1 and q_2 goes only to state q_4, so this will appear in the DFA as state $\{q_4\}$, with an arrow to it on input a. The input b to states q_1 and q_2 leads to state q_3, and thus we add $\{q_3\}$ to the DFA and draw the transition.

The next new state that was created for the DFA is $\{q_4\}$. On input a, q_4 goes to q_1 in the NFA, so we add a transition on a to a new DFA state $\{q_1\}$. On input b, q_4 leads nowhere, so a b transition goes from $\{q_4\}$ to $\emptyset$ in the DFA.

Next, we process $\{q_3\}$. Both inputs a and b must lead to $\emptyset$.

$\{q_1\}$ is the last state that was added to the DFA. Since the only transition from q_1 in the NFA goes to q_4 on a, the DFA goes to the existing state $\{q_4\}$ on a and to $\emptyset$ on b. No more unprocessed DFA states remain.

Finally, a state in the DFA is accepting iff it includes at least one accepting NFA state. The accepting NFA states are q_1 and q_3, so the three accepting DFA states are $\{q_1\}$, $\{q_1, q_2\}$ and $\{q_3\}$, as shown by the double circles in the diagram. The DFA is now complete.

The detailed algorithm is as follows. This is based on the description in Hopcroft, Motwani, and Ullman ([16], pp. 60-66), but it originated from M.O. Rabin and D. Scott [24].

1. On input N, where N is an NFA, to create an equivalent DFA $D = (Q_D, \Sigma, \delta_D, q_{0D}, F_D)$:
2. Set the alphabet Σ for D equal to the original alphabet of N.
3. Create an empty table for the new transition function δ_D, where the columns are the symbols of Σ and the rows will be the new states.
4. Set the DFA start state q_{0D} equal to $\{q_0\}$, where q_0 is the start state of N.
5. Create a queue and add q_{0D} to it.
6. Until the queue is empty (i.e., until no more new states are added to D):
   (a) Get a DFA state p (a set of NFA states) from the queue.
   (b) For each input symbol a in the alphabet:
      i. Find the set of successor states $r = \bigcup_{t \in p} \delta_N(t, a)$, where δ_N is the transition function of NFA N and p is viewed as a collection of NFA states.
         A. If r is not already a state in D, then add it to the queue. (Note that if $r = \emptyset$ because transitions do not exist for input symbol a, then r will become a dead state whose exiting transitions will only lead back to itself.)
      ii. Add this set of states r as an entry to the δ_D table for input a coming from state p, since $\delta_D(p, a) = r$.
   (c) Delete p from the queue.
7. The set of states Q_D in D equals the set of row headers in the δ_D table.
8. Initialize the DFA's set of final states, F_D, to the empty set.
9. Scan through Q_D and add any state containing at least one accepting state of N to F_D.

The time complexity of this algorithm is dominated by the loop in step 6. Let n be the number of states in the original NFA N. Since up to 2^n states may be added to D, and hence to the queue, the loop may be run up to 2^n times. The inner loop in step 6(b) is run once for each symbol in the alphabet, and we assume that the alphabet is of constant size, so this can be disregarded for purposes of calculating complexity. In step 6(b)i, which is essentially also a loop, the NFA transition function δ_N is calculated for every state in the "set" p of NFA states (a single DFA state). This will multiply our complexity by n^2: First, note that p could contain anywhere from 1 to n NFA states. Technically, for each alphabet symbol, p would only include all n states once at most, since p is a distinct subset of the states of N in each iteration of the loop, and there is only one subset containing all n states. Most iterations will contain fewer than n states. Second, for each state in p we have to actually calculate δ_N, and we may

have to follow up to a maximum of n transitions each time. Therefore, taking these two factors of n into account, a loose upper bound on the complexity for the subset construction is $O(n^2 2^n)$. This is terribly slow; however, this does not represent the typical case. Because of the fact that only one state contains all n of the NFA states, this bound will not (quite) be achieved. Moreover, the lazy evaluation may help to reduce the critical exponential factor.

For the sake of simplicity, the algorithms in this discussion generally assume that the NFAs do not contain ε transitions, but it is not difficult to accommodate them. (The JFLAP software does allow ε transitions, but they are typed in using the "!" key. By default, they are labeled as λ transitions, but the setting can be changed so that transitions say ε instead of λ.) For example, in the subset construction pseudocode above, it would only be necessary to make two changes to handle ε. First, set the DFA start state q_{0D} equal to the union of $\{q_0\}$ and the full set of all states reachable from q_0 on the empty string (including all states reachable by following multiple transitions on ε), where q_0 is the start state of N. Second, in the spot where r is specified in the loop, if there are ε transitions exiting any of the states in r, then all of the states reachable on ε (including their successors on ε) must also be added to r. Formally, to make both of these changes, we require the "epsilon-closure" of state q, $ECLOSE(q)$. This set is recursively defined to include all states reachable from q on zero or more ε transitions:

Basis: $q \in ECLOSE(q)$.

Induction: If state $u \in ECLOSE(q)$ and $v \in \delta(u, \varepsilon)$, then $v \in ECLOSE(q)$.

Thus, q_{0D} in our algorithm above simply becomes $ECLOSE(q_0)$. Similarly, to determine r, define a temporary variable $R = \bigcup_{t \in p} \delta_N(t, a)$ and then we get $r = \bigcup_{r_i \in R} ECLOSE(r_i)$.

This does add to the time complexity of the algorithm, since the amount of work to obtain $ECLOSE(q)$ depends not only upon the number of ε transitions exiting q itself, but also upon the number of ε transitions exiting each state which is reached and then recursively checked to see if even more neighbors should be included. In other words, the runtime to determine $ECLOSE(q)$ is impacted by the total number of states which are reached, whether they are immediate neighbors of q or must be reached by increasing the depth of recursion. Clearly, one upper bound on the potential size of $ECLOSE(q)$ is n, the total number of states in the NFA. Another upper bound would be dictated by the total number of ε transitions in the NFA; after visiting q, we cannot possibly add more states to $ECLOSE(q)$ than there are total ε transitions. So the maximum feasible depth of recursion equals the number of ε transitions. In the worst case where each state leads to exactly one brand new state, this would be up to at most n states. When implementing this algorithm for large automata, a lookup table of the states reachable on input ε could speed things along; if δ_N was already represented as a table, just use the ε entries in that table for quick reference. To keep the runtime reasonable, it would also be desirable to stop the recursion

upon encountering a state which was already added to $ECLOSE(q)$ previously. Possible pseudocode to determine $ECLOSE(q)$:

1. On input N, where N is an NFA containing epsilon transitions, and input q, where q is a state in N:
2. Create a set $ECLOSE(q)$ and add q to it.
3. Create a queue and place q on the queue.
4. Until the queue is empty:
   (a) Get a state u from the queue and delete it from the queue.
   (b) Look up the set of states $\delta_N(u, \varepsilon)$ from the transition table. For each state $v \in \delta_N(u, \varepsilon)$:
      i. If $v \notin ECLOSE(q)$: add v to $ECLOSE(q)$ and also to the queue.
5. Output $ECLOSE(q)$.

It is not necessarily feasible to do this within linear $O(n)$ time, despite the transition function lookup table, because each state v which is encountered must be processed; each v must at least be checked to see if it has been encountered before. In the absolute worst case, if every state had an ε transition to every other state, n items would be dequeued, and n states v would be found each time, giving an $O(n^2)$ runtime for this function. The subset construction without ε transitions was estimated to be $O(n^2 2^n)$. We may have to do the $O(n^2)$ epsilon closure up to n times for each loop, so this multiplies the complexity by n^3. Thus, the overall loose bound for the subset construction increases to $O(n^5 2^n)$ if ε transitions are present.

3.2.1 The Reversing Minimization Algorithm

In a chapter on "Regular Languages," Sheng Yu discusses minimization of a DFA using reversal with subset construction [37]. The reversal of input string w is denoted w^R, where w^R consists of the same symbols as w, but in the opposite order. For example, if w is the string 110, then w^R would be 011. If L is the language of strings accepted by DFA A, then $L^R = \{w^R \mid w \in L\}$. Yu states, "An interesting observation is that, for a given DFA, if we construct an NFA that is the reversal of the given DFA and then transform it to a DFA by the standard subset construction technique (constructing only those states that are reachable from the new starting state), then the resulting DFA is a minimum-state DFA." (p. 57) Hing Leung gives a clear, simple version of the same proof for this theorem as follows [22]. The input is a reduced DFA A, (Q, Σ, δ, q_0, F), where "reduced" means that all unreachable states have already been removed. Leung defines $L_q = \{w \mid \delta(q, w) \in F\}$ where $q \in Q$, and specifies that states p and q are "mergeable" (equivalent) if the languages accepted by these states are equivalent, meaning $L_p = L_q$. To get the reverse of A, the

arrows for the transitions are reversed, the initial state becomes the final state, and the final states become initial states. Notice that this is not an ordinary NFA, because there may be multiple initial states. Yu calls it an "NNFA," for "NFA with nondeterministic starting state," p. 15. Clearly, this NNFA will accept the reverse of the strings accepted by A. The final step is to perform the subset construction on the NNFA, leaving out unreachable subsets, to create DFA B. To prove that B is minimal, Leung shows that no two states in B are equivalent, so no more states can be merged together. Specifically, let Q_1 and Q_2 be two states in B. These are actually subsets of states in A. Choose some state $q \in Q_1$ where $q \notin Q_2$. This is always possible because Q_1 and Q_2 are never identical subsets in the subset construction algorithm. Since there are no unreachable states in A, there exists a string w which goes from initial state q_0 to this state q. Since A is a DFA, this is the ONLY state which is reached on input w. Therefore, $w \in L_{Q1}$, but $w \notin L_{Q2}$. Since these languages L_{Q1} and L_{Q2} differ, no two states Q_1 and Q_2 in B are equivalent. Thus, no two states in B can be merged together, and B must be minimal.

Yu describes a commonly known minimization algorithm (originally by Brzozowski), which makes use of this theorem. The algorithm simply performs the process twice: If you reverse DFA A, apply the subset construction to get minimal DFA B as described above, reverse again, and apply the subset construction again, then the final result A' is a minimal DFA equivalent to A. This is because $(w^R)^R = w$. Yu points out that "The algorithm is descriptively very simple. However, the time and space complexities of the algorithm are very high; they are both on the order of 2^n in the worst case, where n is the number of states [in the original DFA]" (p. 58). Because of the subset construction, this is an expensive minimization algorithm. Better options are available. We will see some of these options later as we discuss equivalence-testing algorithms which also minimize DFAs.

3.3 Equivalence-Testing Algorithms

This section describes some of the most significant algorithms for determining the equivalence of finite automata and discusses their complexity. For example, one of these algorithms uses the symmetric difference to check if there are any strings which one automaton accepts and the other does not. Often, minimization algorithms can also be used for purposes of testing if two DFAs recognize the same language. For instance, Hopcroft, Motwani, and Ullman present a "table-filling" algorithm to test if two DFAs are equivalent. Where n is the total number of states in the two DFAs, they show that their algorithm runs in $O(n^4)$ time, or in $O(n^2)$ time if lists of states depending on distinguishable states are tracked and labeled as distinguishable (p. 159). Hopcroft also published an $O(n \lg n)$ algorithm which can be used to minimize or to show equivalence. Moreover, Hopcroft and Karp provide an algorithm which can be used if equivalence is to be tested without doing any work to minimize the automata; it can run in very nearly linear time on n. Each of these algorithms is discussed in more detail below, and the existing JFLAP algorithms are described for purposes of

comparison.

As an interesting side note, Ravikumar and Eisman define the concept of *weak equivalence*, where "two DFA's are weakly equivalent if they both accept the same number of strings of length k for every k."([25], p. 115) They provide an algorithm to determine weak equivalence and suggest that it can be helpful in solving counting problems. Moreover, Watson and Daiciuk [36] suggest that it may be useful to employ a DFA minimization algorithm which provides partial (incremental) results, even it is not run to completion. However, for our purposes, we will focus on proving DFA equivalence, and we will only take shortcuts in cases where the equivalence proof is already known to be complete.

3.3.1 Symmetric Difference with Emptiness Testing

It is well known that the problem of determining whether or not two DFAs accept the same language is decidable. See, e.g., Sipser's chapter on "Decidability," Theorem 4.5, ([29] p. 155): EQ_{DFA} is decidable, where $EQ_{DFA} = \{(A, B) \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$. The proof of this theorem can be used as an algorithm to test equivalence of DFAs. (BFLAP takes this approach for grading students' automata [21]. See the section on BFLAP below.)

The idea is to find out if there are any strings that are accepted by A and not by B. Similarly, check if B accepts any strings which A does not. If the result in both cases is the empty set, then A and B must be equivalent. Otherwise, they cannot be equivalent. To do this, we take the symmetric difference to form DFA C such that

$$L(C) = (L(A) - L(B)) \cup (L(B) - L(A)) \tag{1}$$
$$= (L(A) \cap \overline{L(B)}) \cup (L(B) \cap \overline{L(A)}) \tag{2}$$

where

- The DFA A^C for the complement of $L(A)$, $\overline{L(A)}$, is constructed by swapping the accepting and non-accepting states of DFA A. A^C will then accept everything in Σ^* which A does not, and vice versa.
- The union of two DFAs D_1 and D_2 is formed by taking the Cartesian product of the states in D_1 with the states in D_2, and then adjusting the transition function, start state, etc. so that both DFAs can be simulated simultaneously in a single automaton which will accept $L(D_1) \cup L(D_2)$ (see, e.g., Sipser, p. 46, to be discussed in a moment).
- One way to find the intersection is by using the complement and union; via De Morgan's law, $\overline{L(A) \cap L(B)} = \overline{L(B)} \cup \overline{L(A)}$. Therefore, $L(A) \cap L(B) = (\overline{\overline{L(B)} \cup \overline{L(A)}})$. However, an easier way to get the intersection is to use the same algorithm as for the union (below), but with a restricted set of accepting states.

Then $L(C)$ is tested for emptiness. A and B are equivalent iff C does not recognize any strings. (Otherwise, $L(C)$ is the language of all possible witnesses demonstrating why A and B are not equivalent.)

Here is the procedure for creating the union of two regular languages by simulating both DFAs at the same time (Sipser mentions this method, [29] pp. 45-46). Call the two DFAs D_1 and D_2. Since we are taking the Cartesian product of the states, the number of states in resulting DFA D will be the product of the number of states in D_1 and the number of states in D_2.

1. On input (D_1, D_2), where the DFAs $D_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$ and $D_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$, create $D = D_1 \cup D_2$ as follows:
2. The set of states for D consists of pairs $Q = \{(r_1, r_2) \mid r_1 \in Q_1 \text{ and } r_2 \in Q_2\}$.
3. The alphabet for D is still Σ.
4. For each $(r_1, r_2) \in Q$ and $a \in \Sigma$, the transition function δ becomes $\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a))$.
5. The start state q_0 for D is (q_{01}, q_{02}).
6. The set of accepting states for D is $F = \{(r_1, r_2) \mid r_1 \in F_1 \text{ or } r_2 \in F_2\}$. Iff either of the automata accept a string, it will be accepted by D and hence is included in $L(D)$.

To form the intersection instead of the union, just change the accepting states of D to be only those states in which both r_1 and r_2 are accepting. In other words, $F = \{(r_1, r_2) \mid r_1 \in F_1 \text{ and } r_2 \in F_2\}$ for the intersection.

To test $L(C)$ for emptiness, Sipser uses a commonly known, simple reachability algorithm ([29], p. 154). It begins by marking the start state of C. It then follows the transitions out of each marked state, and tracks and marks any new unmarked states as they are encountered. When no more states can be marked, the algorithm halts. If (and only if) the set of marked states does not include any accepting states, then C represents the empty language, i.e., it does not accept any strings because no accepting states are reachable from the start state. Sipser does not specify how the marked states should be visited or tracked. One way to implement this would be to add newly marked states to a "marked" list, so that we would know which ones were reached so far. Each newly marked state could also be added to the end of a processing queue. They would be removed from the queue one by one as their successor states were checked to see if they still needed to be marked and queued (essentially a breadth-first approach). This way, each state would only need to be considered once, at most. Although all transitions from each state would need to be evaluated, the number of transitions per state is a constant equal to the number of alphabet symbols (we are looking at DFAs here, not NFAs), so the complexity would be $O(n)$ to mark up to n states. This assumes the DFA is represented in a manner that requires only constant time to find the next state; a lookup table for δ would work. It would also only take linear $O(n)$ time to scan the completed list of marked states to determine if any were accepting, or not.

As we will see, JFLAP also uses a reachability algorithm for a different purpose; namely, to detect useless states. Any unreachable states (accepting or not) can safely be eliminated from an automaton since they are not necessary to specify the language which the automaton accepts. This preliminary minimization step can cut down on storage space and/or extraneous calculations. It has the potential to make a noticeable improvement on runtime if it is executed before an expensive algorithm such as the subset construction. Also, for isomorphism checking algorithms, it is important that any unreachable states first be removed. This assures that the DFA is truly minimal, and therefore unique for a given language.

3.3.2 The Table-Filling Algorithm

Students of computer science theory are often familiar with the table-filling algorithm, as it is effective and easy to understand. It can be used both for purposes of minimizing a DFA and for checking equivalence of two DFAs. Hopcroft, Motwani, and Ullman describe it in their *Introduction to Automata Theory, Languages, and Computation*, section 4.4 (pp. 154 - 164).

This algorithm can be used on a single DFA to minimize it. However, to test equivalence, two DFAs are treated almost as if they were a single DFA. The strategy here is to look at all of the states for both DFAs, and to identify states which are distinguishable on some input. As distinguishable states are found, they can be used to identify more distinguishable states: if some input to a pair of states results in a transition to two distinguishable states, then the pair is itself distinguishable. Note that we assume that the names of the states in the two automata are distinct, so that the transition function δ is simply the combination of the transition functions for the two automata. Once the algorithm is completed, if the start states of the two DFAs are still not distinguishable, then these DFAs accept the same language, and thus they are equivalent.

To start with an example, let us test the DFAs in Figures 8 and 9 for equivalence. (We saw these earlier, in Figures 4 and 5.) The first step is to draw a table listing all of the states for both automata, as shown in Figure 10. We only need the lower half of the table since each state only needs to be compared once against each of the other states. All entries where one state is accepting and the other is not are immediately marked with an 'X.' Here, the states q_c and q_g are marked as distinguishable from everything except for each other, since they are the only final states.

Next, we loop through the unmarked pairs of states. We test each input on each member of the pair to see what new pair they will lead to. If they lead to a marked pair on either input 0 or 1, then the original pair is also distinguishable and must be marked. This looping process is repeated until no more pairs can be marked. Starting with (q_a, q_b) on input 0, the diagram shows that q_a goes to q_b and q_b goes to q_c. Pair (q_b, q_c) is already marked, so (q_a, q_b) must now be marked. The next unmarked pair in that column is (q_a, q_d). On input 0, this leads to (q_b, q_d), which is currently unmarked. So we try input 1, and this goes

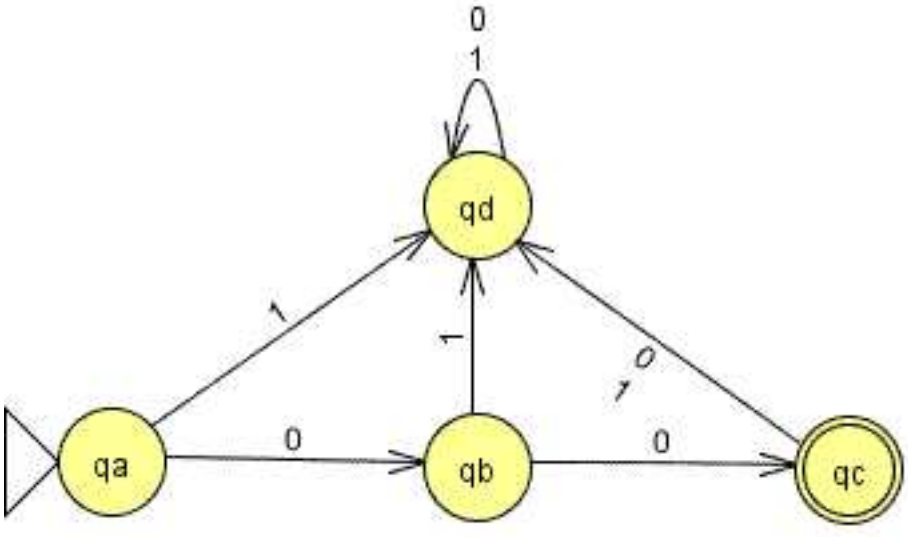

Figure 8: A DFA which accepts only the string 00.

to (q_d, q_d), but q_d is not distinguishable from itself, so we cannot mark (q_a, q_d) at this time, and we move on to the next pair. (q_a, q_e) goes to (q_b, q_f) on input 0 and to (q_d, q_h) on input 1. So far, neither of these are marked. Continuing onwards, when we look at pair (q_a, q_f) and try input 1, (q_d, q_g) is marked, so (q_a, q_f) gets marked. (q_a, q_h) does not get marked at this stage. (q_b, q_d) goes to (q_c, q_d) on input 0, so it is now marked. (q_b, q_e) goes to (q_c, q_f) on 0 and hence is distinguishable. (q_b, q_f) leads to (q_c, q_h) and so is also marked on 0. The reader can verify that the next pairs to be distinguished are: (q_b, q_h), (q_d, q_f), (q_e, q_f), and (q_f, q_h). Then we take a second pass through the table. (q_a, q_d) is now distinguishable on 0 since (q_b, q_d) has been marked. (q_a, q_e) is similarly distinguishable since (q_b, q_f) was marked. If we just wanted to prove that the two DFAs are inequivalent, we could stop the algorithm here, since q_a and q_e are the start states. See Figure 11. Otherwise, we can go on to mark (q_a, q_h), (q_d, q_e), and (q_e, q_h). Then, in the third pass, there are only two remaining pairs. We find that (q_c, q_g) still cannot be marked. (These states always lead to dead states). Since (q_d, q_h) also cannot be marked (they are the dead states and only lead to themselves), nothing else has changed on this pass through the table, and the algorithm is done. Refer to Figure 12.

The pseudocode is as follows:

1. On input (M_1, M_2), where M_1 and M_2 are DFAs, NFAs, and/or regular expressions:

2. Convert M_1 and M_2 to DFA format, if necessary.

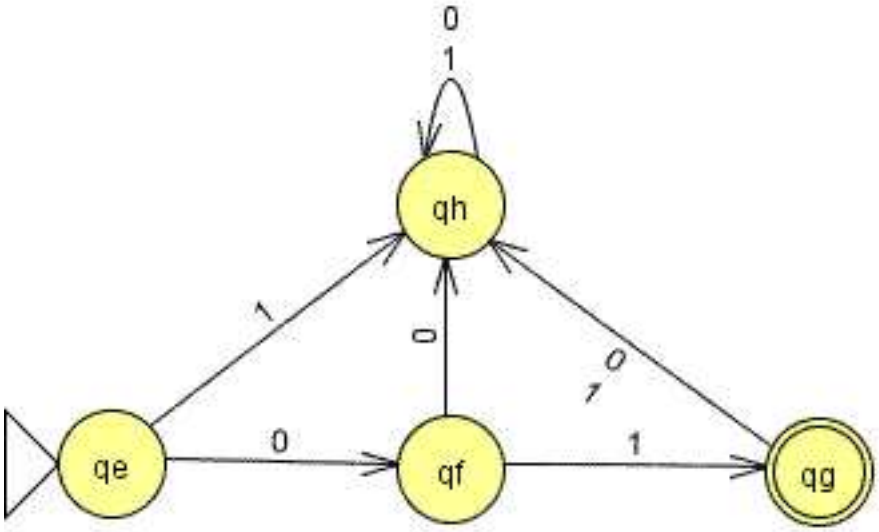

Figure 9: A DFA which accepts only the string 01.

	qa	qb	qc	qd	qe	qf	qg
qb							
qc	X	X					
qd			X				
qe			X				
qf			X				
qg	X	X		X	X	X	
qh			X				X

Figure 10: Initial Table for the Table-Filling Algorithm.

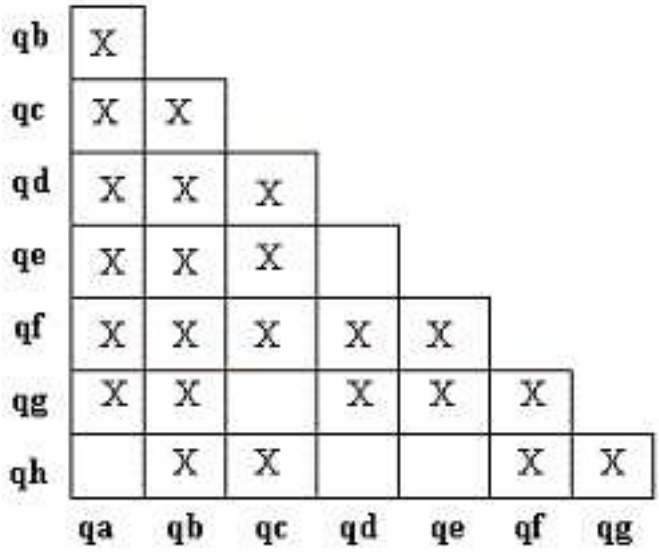

Figure 11: Partially Filled: q_a and q_e are not equivalent.

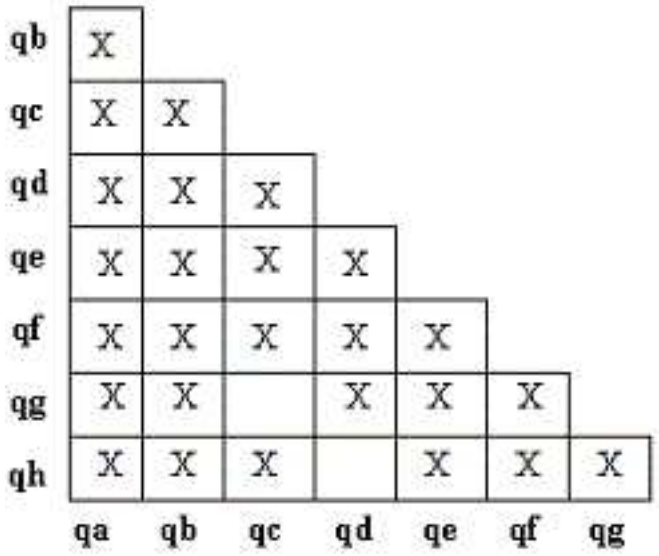

Figure 12: Completed Table for the Table-Filling Algorithm.

3. Construct an $n \times n$ matrix with a row for each state and a column for each state, where n is the sum of the number of states in M_1 and the number of states in M_2. Only the area below the diagonal of the matrix needs to be considered.

4. Mark each entry in this table as distinguishable if one of the states is accepting and the other is not. (This is because on input ε, one state accepts, and the other rejects, so they do not accept the same set of strings.)

5. Loop until no additional pairs of states are marked as distinguishable:

   (a) For each pair of states $\{p, q\}$ that is not yet marked distinguishable:

      i. For each input symbol a_i in the alphabet as long as $\{p, q\}$ is not yet marked distinguishable:

         A. If the pair of states $\{\delta(p, a_i), \delta(q, a_i)\}$ is distinguishable, then mark $\{p, q\}$ as distinguishable.

6. If the start states of the two DFAs are marked as distinguishable, then M_1 and M_2 are not equivalent. Otherwise, they are equivalent.

Hopcroft et al. show that if the inputs are already DFAs, the time complexity of this algorithm is $O(n^4)$, where n is the sum of the number of states in the two DFAs, and the number of input symbols in the alphabet is assumed to be a constant. Specifically, there are $\binom{n}{2} = n(n-1)/2$ pairs of states, so it takes $O(n^2)$ time to loop through each pair on line 5(a) of the algorithm. In each outer loop on line 5, at least one more pair must be marked distinguishable, or the algorithm stops, so no more than $O(n^2)$ outer loops will be performed. The result of the multiplication is $O(n^4)$. This is a maximum because pairs of states will be eliminated as the algorithm continues. The time for step 4 can be disregarded because it is only a single $O(n^2)$ loop.

The authors also show that the run time can be reduced to $O(n^2)$ by listing pairs of states that depend on each other. In other words, we track which pairs of states lead to other pairs of states via the transition function, δ, on a single input symbol. Then we will only have to test each pair once to fill the table.

Going back to rework our previous example of Figures 8 and 9, we begin by creating the lists of predecessors for each pair. For conciseness, pairs without predecessors are omitted below; for instance, there are no predecessors for any pair containing a start state (states q_a or q_e), so they have an empty list. Duplicates need not be saved if the same pair is a predecessor on both inputs 0 and 1.

(q_b, q_c): (q_a, q_b)

(q_b, q_d): (q_a, q_c), (q_a, q_d)

(q_b, q_f): (q_a, q_e)

(q_b, q_h): (q_a, q_f), (q_a, q_g), (q_a, q_h)

(q_c, q_d): (q_b, q_c), (q_b, q_d)

(q_c, q_f): (q_b, q_e)

(q_c, q_h): (q_b, q_f), (q_b, q_g), (q_b, q_h)

(q_d, q_f): (q_e, q_c), (q_e, q_d)

(q_d, q_g): (q_a, q_f), (q_b, q_f), (q_c, q_f), (q_d, q_f)

(q_d, q_h): (q_c, q_f), (q_c, q_g), (q_c, q_h), (q_d, q_f), (q_d, q_g), (q_d, q_h), (q_a, q_e), (q_b, q_e), (q_c, q_e), (q_d, q_e), (q_a, q_g), (q_b, q_g), (q_a, q_h), (q_b, q_h)

(q_f, q_h): (q_e, q_f), (q_e, q_g), (q_e, q_h)

(q_g, q_h): (q_f, q_e), (q_f, q_g), (q_f, q_h)

Then create the table, as before in Figure 10, with accepting/non-accepting pairs immediately marked. However, this time, also add these pairs to a processing queue. The queue should now contain 12 entries:

(q_a, q_c), (q_b, q_c), (q_c, q_d), (q_c, q_e), (q_c, q_f), (q_c, q_h), (q_a, q_g), (q_b, q_g), (q_d, q_g), (q_e, q_g), (q_f, q_g), (q_g, q_h).

The next step is to go through the queue. For each pair in the queue, any unmarked pairs in its list are marked and added to the end of the queue. If we take the queue in the same order as above, (q_a, q_c) has an empty list (no predecessors), so nothing gets marked. (q_b, q_c) is the next item on the queue, and it has the unmarked predecessor (q_a, q_b). Mark (q_a, q_b) and add it to the queue. Continuing with (q_c, q_d), it has two items in its list. (q_b, q_c) has already been distinguished, but (q_b, q_d) has not, so this pair is marked and queued. ... And so forth. This process ends when nothing unprocessed remains on the queue. Although the order of marking will differ from that in the previous algorithm, the end result will still be the same as Figure 12. Since the slot in the table for the start states q_a and q_e is marked, the two DFAs are not equivalent.

The pseudocode for this approach would be as follows:

1. On input (M_1, M_2), where M_1 and M_2 are DFAs, NFAs, and/or regular expressions:
2. Convert M_1 and M_2 to DFA format, if necessary.
3. Create (read-only) lists of predecessor states. Specifically, for each pair of states $\{p, q\}$:
   (a) For each input symbol a_i in the alphabet:
      i. Add $\{p, q\}$ to the list for pair $\{\delta(p, a_i), \delta(q, a_i)\}$, to track the dependency.

4. Construct an $n \times n$ matrix with a row for each state and a column for each state, where n is the sum of the number of states in the two DFAs. Only the area below the diagonal of the matrix needs to be considered.

5. Mark each entry in this table as distinguishable if one of the states is accepting and the other is not. Create a processing queue, and add each distinguishable pair to this queue.

6. While there are pairs in the queue, get the list for the next pair.

   (a) For each pair $\{r, s\}$ on this list not yet marked distinguishable:

      i. Mark $\{r, s\}$ as distinguishable in the table.

      ii. Add $\{r, s\}$ to the queue.

7. If the start states of the two DFAs are marked as distinguishable, then M_1 and M_2 are not equivalent. Otherwise, they are equivalent.

Assuming M_1 and M_2 are already DFAs, the time complexity will be just $O(n^2)$ for the loop in step 3 to create the lists (since there are still $O(n^2)$ pairs of states, and the number of input symbols is assumed constant), plus $O(n^2)$ to find the initial accepting/non-accepting combinations in a single loop in step 5, plus another $O(n^2)$ to process step 6. Note that step 6 is a nested loop, but each pair will only be queued and processed at most once. This results in a sum total polynomial time of $O(n^2)$. The only additional cost versus the previous algorithm is the creation of a set containing up to $\binom{n}{2}$ lists, plus a queue.

As we will see, if the two DFAs do not recognize the same language, it is relatively easy to modify the table-filling algorithm(s) to produce a string which is accepted by one DFA but not by the other.

What if you wanted to use this algorithm on a single DFA to minimize it? Obviously, it would only be necessary to incorporate the states from that particular DFA in the table. Then, by looking through the completed table, it is easy to pick out the groups of states which are not marked as distinguishable. These table-filling algorithms can thus be viewed as "partitioning" algorithms; they essentially partition the states into groups (sets) which have been determined to contain only equivalent states. Minimization would simply require combining each set of states in the partition into a single state. (Any unreachable sets would need to be removed.)

3.3.3 The $O(n^4)$ Table-Filling Algorithm, with Witness

What if a fast equivalence-checking algorithm is desired, but the user would like to have an example of an input string which differentiates the two automata? For instance, what if a student who is using JFLAP needs clarification to understand why two DFAs do not accept the same language? As part of this project, the Hopcroft $n \lg n$ algorithm and a near-linear algorithm (discussed later) were modified and implemented to produce a witness string which is accepted by one DFA, but not by the other. However, for purposes of comparison, let's suggest a way to modify the table-filling algorithms to produce a witness string first.

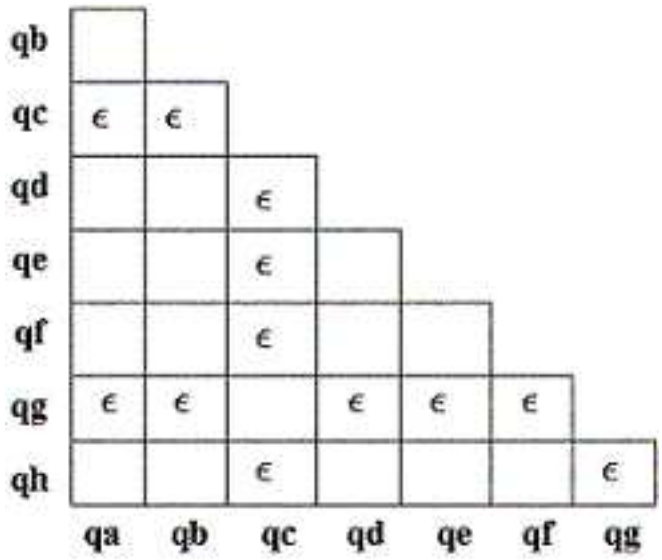

Figure 13: Initial Table for the Table-Filling Algorithm with Witness.

Consider the simplest form of the standard $O(n^4)$ table-filling algorithm. Ordinarily, the table is populated with an 'X' or some similar flag to indicate that a pair of states was found to be distinguishable. However, we can instead track the actual input symbol which was used to distinguish each pair of states. Then, after the table has been completed, we can work backwards from the table using the transition function δ to construct the witness.

Figures 13 through 15 show what our previous example from Figures 4 and 5 would look like with the witness incorporated. In Figure 13, the table is initialized with ε in the entries where one state is accepting and the other is not; these states are distinguishable on the empty string. Next, we loop through the unmarked pairs of states until no more pairs can be marked, as before. The first pair, (q_a, q_b), leads to marked pair (q_b, q_c) on input 0, so it is now marked with a 0 in the table. Continuing down that column, (q_a, q_d) cannot be marked yet, nor can (q_a, q_e). The next pair is (q_a, q_f), which goes to marked states (q_d, q_g) on input 1, so a 1 is added to the table, and so forth. Figure 14 shows what the result looks like if we pause the algorithm when the start states are shown to be inequivalent. Figure 15 displays the completed table.

The table can then be used to formulate a witness string. Begin with the start states, (q_a, q_e). The corresponding symbol in the table is 0, so that is the beginning of the witness. The next state pair is calculated as $(\delta(q_a, 0), \delta(q_e, 0))$, which is (q_b, q_f). The table entry for (q_b, q_f) is 0, so we append this to the end of the witness, which now becomes 00. Compute $(\delta(q_b, 0), \delta(q_f, 0))$ to obtain (q_c, q_h). Since q_c is a final state and q_h is not, they are distinguishable on the

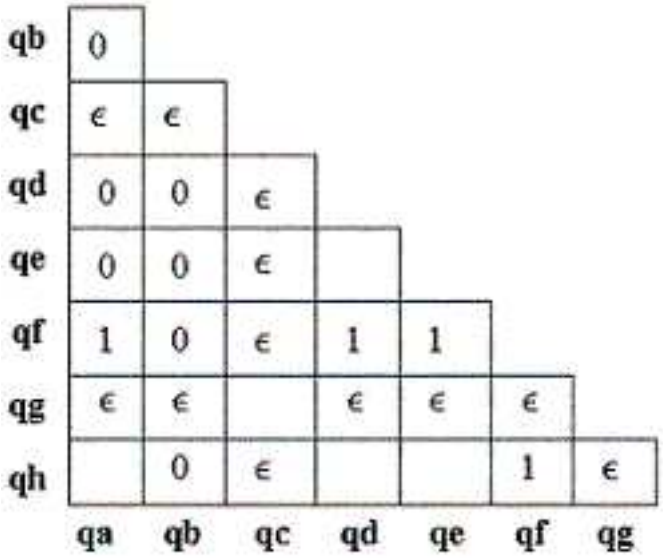

Figure 14: Partially Filled: q_a and q_e are not equivalent.

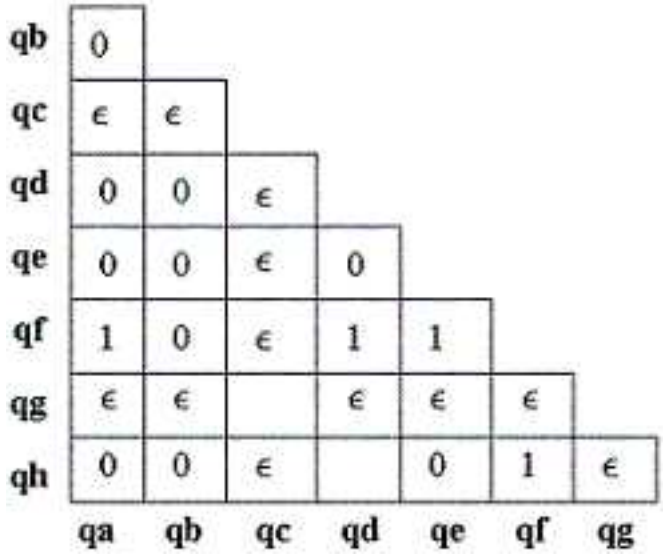

Figure 15: Completed Table for the Table-Filling Algorithm with Witness.

empty string, and the witness 00 is complete. One automaton accepts 00, and the other does not. Notice that 01 would also be an acceptable solution for this example, depending on the order in which the inequivalent states are found and marked.

The pseudocode for this new algorithm would be:

1. On input (M_1, M_2), where M_1 and M_2 are DFAs, NFAs, and/or regular expressions:
2. Convert M_1 and M_2 to DFA format, if necessary.
3. Construct an $n \times n$ matrix with a row for each state and a column for each state, where n is the sum of the number of states in M_1 and the number of states in M_2. Only the area below the diagonal of the matrix needs to be considered.
4. Mark each entry in this table with an ε if one of the states is accepting and the other is not. (This is because on input ε, one state accepts, and the other rejects, so they do not accept the same set of strings.)
5. Loop until no additional pairs of states are marked as distinguishable:
   (a) For each pair of states $\{p, q\}$ that is not yet marked distinguishable:
      i. For each input symbol a_i in the alphabet as long as $\{p, q\}$ is not yet marked distinguishable:
         A. If the pair of states $\{\delta(p, a_i), \delta(q, a_i)\}$ is distinguishable, then mark $\{p, q\}$ as distinguishable by entering the symbol a_i in the table.
6. If the start states of the two DFAs are not marked as distinguishable, then M_1 and M_2 are equivalent. Output "equivalent, no witness" and halt.
7. Otherwise, the automata are not equivalent. To form the witness string,
   (a) Initialize the witness to the empty string.
   (b) Look up the table entry a for the two start states. Call these states p_w and q_w.
   (c) Until one of the states p_w and q_w is a final state and the other is not:
      i. Append the symbol a to the end of the witness.
      ii. Set $p_w = \delta(p_w, a)$ and $q_w = \delta(q_w, a)$.
      iii. Set a to the table entry for the new p_w and q_w.
   (d) Output the witness.

The runtime is still $O(n^4)$ for table-filling. We are constructing exactly the same table as before, just with different symbols as entries. (To save time, the algorithm could stop filling the table as soon as it completes an entry for the two start states.) However, as an additional procedure, the witness must now

be formed. Since the table is of size $O(n^2)$, is the time to create the witness after that $O(n^2)$ or smaller? Or is it possible to wind up in an infinite loop? Notice that this algorithm could have symbols entered in any order – and unlike the modified Hopcroft witness algorithm discussed below, we may encounter the same pair $\{p, q\}$ more than once as we fill the table – but this will not cause a problem. Observe that only the first symbol will be saved in the table for any given pair of states. To demonstrate that it can be used to construct a valid, non-infinite witness string, recall the starting point to distinguish states: we identify and mark states which are accepting vs. those which are non-accepting. As the program continues, no symbol is entered in the table unless transitions on that symbol were found to lead to an accepting/non-accepting pair of states. Therefore, when the witness is built by tracing these symbols in reverse via the transition function δ, it must ultimately lead back to an accepting/non-accepting pair. This means that the current string is accepted by one DFA, but not by the other. Therefore, the witness string is complete, and the program will halt. The runtime to produce the witness will not be longer than the length of the string, and since there are $O(n^2)$ entries in the table and we do not repeat configurations (because that would imply an infinite loop), $O(n^2)$ is definitely a limit on the runtime. However, this can be tightened up to $O(n_1 * n_2)$ rather than $O(n^2)$, where n_1 is the number of states in the first DFA, n_2 is the number of states in the second, and n is their sum. The reason for this is that the table entries for states from the same DFA will be ignored in the loop which produces the witness. It calculates $\delta(p_w, a)$ and $\delta(q_w, a)$ only for cases where p_w is from one DFA and q_w is from the other.

Overall, the total runtime is therefore still $O(n^4)$. Moreover, the amount of storage space remains exactly the same as the original table-filling algorithm because it uses exactly the same table structure. It may be useful to save the witness as an additional string, but even that is not strictly necessary because the symbols in the string could be outputted as they are encountered in the final loop. In the next section, let us try to speed up the algorithm.

3.3.4 The Faster $O(n^2)$ Table-Filling Algorithm, with Witness

If we are willing to use a little more storage space, the $O(n^2)$ table-filling algorithm can also be tweaked to create a witness. The idea is the same as the basic $O(n^2)$ algorithm, but the lists of predecessor states must now incorporate the distinguishing symbol a_i as they are saved. The procedure to create the witness itself is identical to the procedure in the previous section. This yields a total runtime of $O(n^2)$ plus $O(n_1 * n_2)$, or just $O(n^2)$. (Again, n_1 is the number of states in the first DFA, n_2 is the number of states in the second, and n is their sum.) This is how the algorithm now looks:

1. On input (M_1, M_2), where M_1 and M_2 are DFAs, NFAs, and/or regular expressions:

2. Convert M_1 and M_2 to DFA format, if necessary.

3. Create (read-only) lists of predecessor states. Specifically, for each pair of states $\{p, q\}$:
   (a) For each input symbol a_i in the alphabet:
      i. Add $\{p, q, a_i\}$ to the list for pair $\{\delta(p, a_i), \delta(q, a_i)\}$, to track the dependency.
4. Construct an $n \times n$ matrix with a row for each state and a column for each state, where n is the sum of the number of states in the two DFAs. Only the area below the diagonal of the matrix needs to be considered.
5. Mark each entry in this table with an ε if one of the states is accepting and the other is not. Create a processing queue, and add each distinguishable pair of states to this queue.
6. While there are pairs in the queue, get the predecessor list for the next pair.
   (a) For each item $\{r, s, a\}$ on this predecessor list, if $\{r, s\}$ is not yet marked distinguishable:
      i. Mark $\{r, s\}$ as distinguishable by entering the symbol a in the table.
      ii. Add $\{r, s\}$ to the queue.
7. If the start states of the two DFAs are not marked as distinguishable, then M_1 and M_2 are equivalent. Output "equivalent, no witness" and halt.
8. Otherwise, the automata are not equivalent. To form the witness string,
   (a) Initialize the witness to the empty string.
   (b) Look up the table entry a for the two start states. Call these states p_w and q_w.
   (c) Until one of the states p_w and q_w is a final state and the other is not:
      i. Append the symbol a to the end of the witness.
      ii. Set $p_w = \delta(p_w, a)$ and $q_w = \delta(q_w, a)$.
      iii. Set a to the table entry for the new p_w and q_w.
   (d) Output the witness.

A simple shortcut in either of the table-filling algorithms would be to break out of the marking loop as soon as the pair of start states was marked. This would not guarantee a shorter runtime, but in many cases, it would help. We don't need to worry about the empty slots in the table because they will not be necessary to form the witness; we already have the distinguishing symbols we need to create the string. Notice that the witness produced by either of the table-filling algorithms may not necessarily be the shortest possible candidate. Before we consider exactly how long this witness string will be, let us look at an alternative to table-filling, for purposes of comparison. The string length will be discussed in more detail after modifying the Hopcroft algorithm.

3.3.5 The $n \lg n$ Hopcroft Algorithm

Hopcroft presented an algorithm in 1971 which is still one of the fastest worst-case approaches to minimizing automata and/or demonstrating equivalence [15]. It runs in time $O(n \lg n)$, where n is the total number of states under consideration. (More specifically, the runtime is $O(cn \lg n)$, where c is equal to the number of symbols in the alphabet. Normally, c is considered to be a constant, and thus disregarded in big-O notation. However, note that experimental evidence [1] suggests that different algorithms may perform better in the average case for different kinds of DFAs.) This Hopcroft algorithm systematically partitions the states into groups which are distinguishable from each other until no more partitioning is possible. One of the strategies to cut down on the amount of work required is to use an inverse transition function table, δ^{-1}, to make it easy to look up predecessor states. This differs from the predecessor lists in the preceding $O(n^2)$ table-filling algorithm because it goes by individual states, rather than by pairs of states, and the input symbols are tracked by default.

Hopcroft's explanation of this complicated algorithm is quite brief, so for a step-by-step introduction to how the pieces fit together, Gries's article [9], "Describing an Algorithm by Hopcroft," may be helpful. As Gries says, "a structured approach to presenting an algorithm seems to be longer and requires more discussion than the conventional way. If the reader wishes to complain about this, he is challenged to first read Hopcroft's original paper and see whether he can understand it easily. The advantage of our approach will then be clear" (p. 1). Gries also fills in some of the gaps in Hopcroft's runtime analysis.

Before we look at the formal pseudocode to test equivalence, let us return to the example of Figures 8 and 9 to see how the process works. (The reader may wish to peek ahead to the pseudocode, as this is a complex algorithm.) To begin, the inverse transition function δ^{-1} for these two DFAs is easily found by inspection of the diagrams; follow the arrows in reverse to construct the following lists:

State	Input 0	Input 1
q_a	$\emptyset$	$\emptyset$
q_b	q_a	$\emptyset$
q_c	q_b	$\emptyset$
q_d	q_c, q_d	q_a, q_b, q_c, q_d
q_e	$\emptyset$	$\emptyset$
q_f	q_e	$\emptyset$
q_g	$\emptyset$	q_f
q_h	q_f, q_g, q_h	q_e, q_g, q_h

Next, create the initial partition. It consists of two sets, or "blocks." The set of accepting states is $B_1 = \{q_c, q_g\}$. The remainder of the states are non-accepting, so the second set is $B_2 = \{q_a, q_b, q_d, q_e, q_f, q_h\}$. Clearly, these two sets are distinguishable from each other (on the empty string). Then, for each

input symbol a and each block, assemble the set of states from that block which has predecessors on that input symbol. To do this, check for a non-null δ^{-1} value. Call this set $a_\alpha(i)$, where i is the block number, and the subscript α indicates the input symbol a for the sake of clarity. For block B_1 on input 0, the inverse transition function on state q_c leads to another state (which happens to be q_b), but state q_g has a null outcome, so $a_0(1) = \{q_c\}$. For block B_1 on input 1, q_c has a null result, but q_g does not, so $a_1(1) = \{q_g\}$. For B_2, input 0 gives $a_0(2) = \{q_b, q_d, q_f, q_h\}$, and input 1 gives the set $a_1(2) = \{q_d, q_h\}$. The goal is to break the blocks into smaller pieces, based on these sets. For each symbol a in the alphabet, pick the smaller set from one of the two blocks. On input 0, B_1 produced a set $a_0(1)$ of size 1, but B_2 produced a set $a_0(2)$ of size 4. Store the block number for the smaller result, block number 1 for input 0, on a processing list $L(a)$ for that input symbol; $L_0(a) = \{1\}$. Strictly speaking, Hopcroft does not treat the processing lists $L(a)$ as lists or queues, but actually as sets which store block numbers in no particular order. They can be implemented as HashSets or Vectors in Java, as they simply store integers (block numbers). However, it is helpful to think of these sets $L(a)$ as "to-do lists," or a collection of items to be processed. When adding to these "lists," we always choose the number of the block with the smaller size $\|a(i)\|$; it ultimately reduces the amount of work to be done because less transitions will be involved. Continuing our example on input 1, B_1 produced a set $a_1(1)$ of size 1, whereas B_2 had a larger set $a_1(2)$ of size 2. Choose the block with the smaller set, and store its block number on the to-do list, so $L_1(a) = \{1\}$. To complete the initialization phase of this algorithm, set a counter k equal to one more than the current quantity of blocks, so $k = 3$.

We are now ready to loop through the data to produce more distinguishable blocks. The looping continues until the sets $L(a)$ for all input symbols are empty. Start with the input symbol of 1. Take the block number $i = 1$ from the to-do list $L_1(a)$, and delete it. (Now $L_1(a) = \emptyset$.) For this block number, $a_1(1)$ was found to be $\{q_g\}$. Look up each state in this set in the δ^{-1} table to get the set of predecessors P on the input symbol; this gives us only one predecessor, as $P = \delta^{-1}(q_g, 1) = \{q_f\}$. For each of these predecessor states, we need to find the block(s) j where they reside. For each such block B_j, we need to put the predecessors into a new block B'_j. In other words, find $B'_j = P \cap B_j$. Here, state q_f is located in block 2. $B_2 = \{q_a, q_b, q_d, q_e, q_f, q_h\}$ can be split. B'_2 is the intersection of B_2 and P, or simply $\{q_f\}$. So we create a new block $B_k = B_3 = B_2 - B'_2 = \{q_a, q_b, q_d, q_e, q_h\}$. The key point is that we have just split a block into two new, distinguishable blocks of states. How do we know this is correct? Recall that states are considered distinguishable when the same result (accept/reject) is not necessarily obtained from both states on the same input string. If two states on some input a lead to two distinguishable states, or even to different blocks that are known to be distinguishable, then they are themselves distinguishable. In this case, for input symbol 1, the states in block B_2 were tested to see if they lead to block number 1 or not. State q_f will lead to block 1 because $\delta(q_f, 1) = q_g$, and $q_g \in B_1$. However, the other states in B_2 do not lead to block 1 on input 1. For example, $\delta(q_a, 1) = q_d$, and $q_d \notin B_1$.

Therefore, we have correctly split block B_2. Observe that the split, if there is a split, will always yield exactly two new blocks because we only separate states that lead to block i from those which do not go to block i on input symbol a. We don't care about the destination block unless it is block number i. Each block is distinguishable from every other block, so this always works (B_i is distinguishable from B_m for all $m \neq i$).

Continuing the algorithm, update $B_2 = B_2' = \{q_f\}$. Since the partition has changed, $a(j)$ and $a(k)$ need to be updated (or created) for each symbol in the alphabet. Currently, $j = 2$ and $k = 3$. For symbol 0, the δ^{-1} table has values for $a_0(j) = a_0(2) = \{q_f\}$ and $a_0(k) = a_0(3) = \{q_b, q_d, q_h\}$. For symbol 1, $a_1(2) = \emptyset$ because q_f has no predecessors on input 1, and $a_1(3) = \{q_d, q_h\}$.

Now the to-do lists $L(a)$ must also be updated with information from the new and changed blocks. Again, we want the smaller non-empty set $a(i)$. For symbol 0, $a_0(2)$ is smaller than $a_0(3)$, so we add block 2 to the to-do list; our new to-do list for symbol 0 is $L_0(a) = \{1, 2\}$. For symbol 1, $a_1(2)$ is empty. Therefore, we must add block 3 to the set of items to be processed. Since it was previously empty, $L_1(a) = \{3\}$. Increment k to 4, the next available block number, and we are done with that iteration of the loop. The blocks in the partition are currently:

$B_1 = \{q_c, q_g\}$

$B_2 = \{q_f\}$

$B_3 = \{q_a, q_b, q_d, q_e, q_h\}$

To partition again: Choose a block number from a to-do list $L(a)$, and repeat the looping process as described in the two previous paragraphs.

In one possible sequence of events, selecting symbol 0 will eventually split B_3 so that the partition becomes:

$B_1 = \{q_c, q_g\}$

$B_2 = \{q_f\}$

$B_3 = \{q_b\}$

$B_4 = \{q_a, q_d, q_e, q_h\}$

Later, a split on symbol 0 gives:

$B_1 = \{q_c, q_g\}$

$B_2 = \{q_f\}$

$B_3 = \{q_b\}$

$B_4 = \{q_e\}$

$B_5 = \{q_a, q_d, q_h\}$

As a shortcut, if we wish, we can stop here. Since the two start states q_a and q_e are not in the same block, the two automata cannot be equivalent. (For our purposes, this is sufficent. We are not minimizing a single automaton, but rather determining equivalence.)

The pseudocode for the algorithm:

1. On input (M_1, M_2), where M_1 and M_2 are DFAs:
2. Let S be the set of states in both automata. For all states $s \in S$ and all $a \in \Sigma$, construct the inverse transition function $\delta^{-1}(s, a) = \{t \mid \delta(t, a) = s\}$. (Stored as a table of lists of states.)
3. Create two initial partition blocks: Set $B_1 = F$, where F is the set of final (accepting) states in both automata. Set $B_2 = S - F$.
4. For each $a \in \Sigma$, for $1 \leq i \leq 2$, let $a(i) = \{s \mid s \in B_i \text{ and } \delta^{-1}(s, a) \neq \emptyset\}$. This is the set of states in block number i which has predecessors via input a.
5. Let k be a counter for the next unused block number. Set $k = 3$.
6. For all $a \in \Sigma$, pick the smaller set and put its block number on a to-do list (unordered set) $L(a)$ to be processed: Let $L(a) = \{1\}$ if $\|a(1)\| \leq \|a(2)\|$. Otherwise, $L(a) = \{2\}$.
7. For each $a \in \Sigma$ and $i \in L(a)$, until $L(a) = \emptyset$ for all a:
   (a) Delete i from $L(a)$.
   (b) Identify the blocks which can be split. For each state $s \in a(i)$:
      i. Use the δ^{-1} table to look up state s. The result is the set of predecessor states. For each predecessor state t:
      ii. Find the block where t resides, and call this block number j.
      iii. Add j to a list of blocks to split, if it's not already listed.
      iv. Add t to a set of states B'_j for that block. (Create the set if needed.)
   (c) For all j in the list of blocks to split (these are the blocks where $\exists t \in B_j$ with $\delta(t, a) \in a(i)$):
      i. If $\|B'_j\| < \|B_j\|$, split B_j into two distinguishable sets:
         A. Let $B_k = B_j - B'_j$.
         B. Set $B_j = B'_j$.
         C. For each $a \in \Sigma$, construct $a(j)$ and $a(k)$.
         D. For each $a \in \Sigma$, update the list of items to process. To minimize the number of transitions to process, we want the smaller set, as long as it was changed. Therefore, let $L(a) = L(a) \cup \{j\}$ if $j \notin L(a)$ and $0 < \|a(j)\| \leq \|a(k)\|$. Otherwise, $L(a) = L(a) \cup \{k\}$.

E. $k = k + 1$.

8. If the start states of the two automata are in the same block, then they are equivalent, so M_1 and M_2 are equivalent. Otherwise, the automata are not equivalent.

Note that the input must be restricted to DFAs for the partitioning steps to work. If NFAs were allowed, the sets could not be split cleanly. There might be overlap where a transition from a single state on a single symbol a could (eventually) lead to both final and nonfinal states. As Hopcroft says, "Blocks are refined ... only when successor states on a given input have previously been shown to be inequivalent." ([15], p. 3)

Hopcroft used linked lists, along with vectors indicating what was or wasn't in the lists, so that the time required to add, delete, or find an item in a list would be fixed. None of the initialization steps (1 through 6) takes longer than $O(n)$. Hopcroft indicates that the nested loop structure in step 7 is executed for $O(n)$ iterations. The bottom line is that there are n states, and the partition cannot be broken into more blocks than states. Significant work is only performed when a block can actually be split. The work involved in each iteration is $O(\lg n)$ because "once we have partitioned on a block and an input symbol, we need never partition on that block and input symbol again until the block is split and then we need only partition on one of the two subblocks ... the time needed to partition on a block is proportional to the transitions into the block and ... we can always select the half with fewer transitions."(p. 2) In other words, since $L(a)$ always tracks the smaller new set $a(i)$, the new $a(i)$ to be processed will be at most half of the size of the old one. This means that the number of states remaining to process for the input symbol and new subblock is at least halved after each iteration, yielding a logarithmic amount of total work. Thus, the total time for all iterations of the nested loop is $O(n \lg n)$. Refer to Gries for an in-depth analysis, as it is beyond the scope of this project. (Note that Hopcroft's pseudocode as originally presented looks at first glance like it is actually $O(n^2)$ because his inner loop appears to process every single block with each iteration of the outer loop. However, the fine print states that this is not the case.) The final comparison of the start states (step 8, if we do not use this as a shortcut condition) would only require checking less than n blocks to determine where these states are located. Therefore, the total run time is $O(n \lg n)$.

Since the time when Hopcroft formulated this well-known algorithm, several researchers have worked to define more intuitive approaches while maintaining $O(n \lg n)$ complexity. For example, see Blum [2] or Knuutila [20]. In 1994, Watson compiled a complete taxonomy of minimization algorithms [35].

In the Java implementation written for this project, the Hopcroft algorithm was used. NFAs are legal as input because they are first converted to DFAs via existing JFLAP subset construction code. Naturally, this slows things down. Even though NFAs are allowed, note that the two automata drawn in JFLAP must use the same alphabet. Each symbol must be used at least once in both models. Otherwise, an error message is displayed to state that the two automata

cannot be compared. The reason for this is that if the new code is run with two different alphabets, a Java NullPointerException will result when the program attempts to look up a state transition on a nonexistent alphabet symbol. To get around this problem, the JFLAP user may revise their NFA diagram to include a dead trap state. Unused symbols would need to transition from some state(s) to this dead state.

3.3.6 The Hopcroft Algorithm, with Witness

To add a witness to the Hopcroft $n \lg n$ algorithm, we follow the spirit of the modifications which were made to the table-filling algorithms. The key is to track the input symbols as they are used to distinguish states. For the modified Hopcroft algorithm, this must be done at least up to the stage where the two start states are placed into different blocks. At the beginning of the program, the first two blocks are initialized. One contains accepting states, and the other contains non-accepting states. Clearly, the two automata are distinguishable on the empty string if the start state from one automaton is accepting, but the start state from the other automaton is not. In this case, the empty string would be the witness string, and execution can stop immediately.

Otherwise, we need to create a string. The $n \lg n$ partitioning algorithm uses the inverse transition function, so by using the standard transition function, we can follow the algorithm backwards. To do this, it is necessary to track the input symbols that are used to distinguish the states from each other as the program executes. To reduce the amount of overhead (space), it is sufficient to track only those symbols which distinguished the states in one automaton from those in the other automaton. (We are not interested in whether or not two states in the same automaton are equivalent.) One way of implementing this would be to create a table where the rows are the states in one automaton, and the columns are the states in the other. As a pair of states p and q is split into two different blocks, the symbol used would be stored in the table. Once the start states are separated (distinguished), the basic partitioning algorithm can stop, and the witness string can be formed by calculating the transition function δ as follows: we begin by going to the table and looking up the symbol that distinguished the start states, and we store that symbol as the first character of the witness string. Then we calculate δ on each of the start states (with that symbol) to get the output states. Next, we check the two output states, and determine if one is accepting and the other is not. As long as that is not the case, we continue calculating δ on each pair of output states using the symbol from the table which distinguished that pair, and append that symbol to the end of the witness. When one state accepts and the other does not, the resulting output is a string which one automaton accepts and the other does not. That is the witness.

See the table in Figure 16 for the symbols used to distinguish states in the example of the previous section. The table shows only the data up to the point where the start states were distinguished. According to the table, 0 was the symbol used to partition the start states q_a and q_e into separate blocks. Thus,

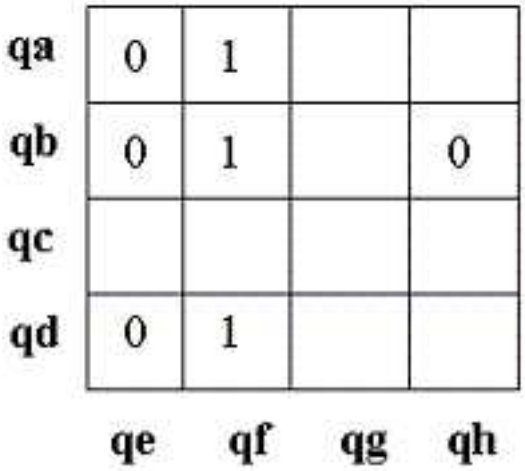

Figure 16: Partial *SYMBOLS* Table for Witness in Modified Hopcroft Approach.

the witness begins with 0. Then $(\delta(q_a, 0), \delta(q_e, 0))$ gives the next pair, (q_b, q_f). From the table, this pair of states was split into separate blocks on input 1, giving witness 01. Finally, $(\delta(q_b, 1), \delta(q_f, 1)) = (q_d, q_g)$, which is a non-final/final pair, so the witness 01 is complete.

The new algorithm becomes the following:

1. On input (M_1, M_2), where M_1 and M_2 are DFAs:
2. Construct an empty table *SYMBOLS* where the columns are the states of M_1 and the rows are the states of M_2. This will store the distinguishing alphabet symbols to be used to form the witness string if M_1 and M_2 are found to be inequivalent.
3. Let S be the set of states in both automata. For all states $s \in S$ and all $a \in \Sigma$, construct the inverse transition function $\delta^{-1}(s, a) = \{t \mid \delta(t, a) = s\}$. (Stored as a table of lists of states.)
4. Create two initial partition blocks: Set $B_1 = F$, where F is the set of final (accepting) states in both automata. Set $B_2 = S - F$.
5. For each $a \in \Sigma$, for $1 \leq i \leq 2$, let $a(i) = \{s \mid s \in B_i \text{ and } \delta^{-1}(s, a) \neq \emptyset\}$. This is the set of states in block number i which has predecessors via input a.
6. Let k be a counter for the next unused block number. Set $k = 3$.
7. For all $a \in \Sigma$, pick the smaller set and put its block number on a to-do list (unordered set) $L(a)$ to be processed: Let $L(a) = \{1\}$ if $\|a(1)\| \leq \|a(2)\|$. Otherwise, $L(a) = \{2\}$.

8. For each $a \in \Sigma$ and $i \in L(a)$, until $L(a) = \emptyset$ for all a:
   (a) Delete i from $L(a)$.
   (b) Identify the blocks which can be split. For each state $s \in a(i)$:
      i. Use the δ^{-1} table to look up state s. The result is the set of predecessor states. For each predecessor state t:
      ii. Find the block where t resides, and call this block number j.
      iii. Add j to a list of blocks to split, if it's not already listed.
      iv. Add t to a set of states B'_j for that block. (Create the set if needed.)
   (c) For all j in the list of blocks to split (these are the blocks where $\exists t \in B_j$ with $\delta(t, a) \in a(i)$):
      i. If $\|B'_j\| < \|B_j\|$, split B_j into two distinguishable sets:
         A. Let $B_k = B_j - B'_j$.
         B. Set $B_j = B'_j$.
         C. Store data for the witness. For each state $t \in B'_j$ and for each state $u \in B_k$: save the current symbol a as the *SYMBOLS* table entry for states (t, u) if t and u are not in the same automaton.
         D. If the start states for the two automata are now in different blocks, M_1 and M_2 are not equivalent. Exit this nested loop and go to the last step below to output the witness.
      ii. For each $a \in \Sigma$, construct $a(j)$ and $a(k)$.
      iii. For each $a \in \Sigma$, update the list of items to process. We want the smaller set, as long as it was changed. Therefore, let $L(a) = L(a) \cup \{j\}$ if $j \notin L(a)$ and $0 < \|a(j)\| \leq \|a(k)\|$. Otherwise, $L(a) = L(a) \cup \{k\}$.
      iv. $k = k + 1$.

9. If the start states of the two automata are in the same block, then these states are equivalent, so M_1 and M_2 are equivalent. Output "equivalent, no witness" and halt.

10. Otherwise, the automata are not equivalent. To form the witness string,
    (a) Initialize the witness to the empty string.
    (b) Look up the *SYMBOLS* table entry a_s for the two start states. Call these states p and q.
    (c) Until one of the states p and q is a final state and the other is not:
       i. Append the symbol a_s to the end of the witness.
       ii. Set $p = \delta(p, a_s)$ and $q = \delta(q, a_s)$.
       iii. Set a_s to the *SYMBOLS* table entry for the new p and q.
    (d) Output the witness.

Since we need to store the *SYMBOLS* table, this algorithm uses a bit more space than the ordinary Hopcroft approach. Let n_1 be the number of states in the first automaton, and let n_2 be the number of states in the second. The table is then of size $(n_1 * n_2)$. If the table must be initialized when the program starts, this would take time $O(n_1 * n_2)$. The initialization time could potentially be reduced by using a structure such as a hashtable or a Java HashMap. However, the time complexity of the main algorithm is still no longer quite $n \lg n$. To fill the *SYMBOLS* table (or other structure) as the program runs (step 8.c.1.C), a nested mini-loop must mark each of the states in the new block as being distinguishable from each of the states in the revised block (while encountering and skipping over pairs of states in the same automaton). In other words, the pairs of newly distinguished states (t, u) must be tracked as distinguished when they are put into separate blocks. Assume n is the total quantity of states in both automata together. Each pair of states will only be encountered at most once. Thus, in the worst case, each state might be checked against every other state during the course of the algorithm, requiring roughly $O(n^2)$ time. This can be viewed as an "additional" step to the original algorithm, as it is only executed as needed and not for every run of the outer loop. Specifically, the $O(n^2)$ work involved is NOT multiplied by the $n \lg n$ work for the original loop which contains it. Rather, it is just added to it. The *SYMBOLS* table is only updated when partitioning actually occurs, and this process is never executed more than once for any pair of states. Once a pair has been split into different blocks, that pair of states is not encountered again. Thus, the runtime for this algorithm with data tracking for the witness is $O(n_1 * n_2)$ plus $O(n \lg n)$ plus $O(n^2)$, for a total of $O(n^2)$.

In reality, you can't quite reach the n^2 worst case. The algorithm begins with at least 2 separate blocks (unless everything is equivalent, in which case, we're already done). We don't track the pairs where each state originates from a completely different block. This is certainly no worse timewise than the basic table-filling algorithm, and possibly better, as long as it is implemented carefully. (We will look at the step 10 for witness creation in a moment.) The actual runtime may be much less than $O(n^2)$ if the algorithm is halted early and a witness is found before the entire *SYMBOLS* table is completed.

For purposes of this project, a Java HashMap was used for *SYMBOLS* rather than a table or array. This meant that a little space was saved, as it was unnecessary to reserve space for, or even to initialize, empty entries. (The impact is insignificant for small automata.) It also reduced the time to access entries. States have no inherent numbering system, and it is undesirable to loop through all of them to find the correct pair, but lookup of a distinguished state pair object (key) can be done quickly in a HashMap to find the alphabet symbol (value). This improves performance. The reader may wish to see Dietzfelbinger et al. for details regarding the design and analysis of efficient randomized hashtables to be used in a "dynamic situation, wherein membership queries are processed in constant worst case time, insertions and deletions are processed in constant amortized expected time and the storage used at any time is proportional to the number of elements currently stored in the table." ([6], p. 524)

Another factor in the implementation for this project is that JFLAP stores transitions individually, not as part of a transition function. JFLAP was designed more with graph algorithms in mind. Therefore, a transition function δ was stored on the fly for this algorithm as the inverse transition function was created. This did not have much time impact, and it does not take much space to store δ for a small DFA.

Note that there is more than one possible witness, depending on the order in which states are processed during the partitioning portion of the algorithm. Hopcroft does not specify how to choose the next symbol/state; any order will do. Since JFLAP stores several classes, including Transitions, in HashSets, and the partitioning portion of the algorithm does not specify the order in which sets are processed, this means that different results may be obtained if the algorithm is run more than once. The HashSet JavaDoc [11] explains that class java.util.HashSet only takes constant time to find a given element, but it "makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time." For instance, in the code written for this project, if the two automata accept 00 and 01 respectively, EquivalenceNlgNWitness.java will output either the witness 01 or the witness 00.

This can even have an impact in terms of witness string length. Neither the Hopcroft $n \lg n$ algorithm nor the table-filling algorithm specifies the exact order in which pairs of states $\{p, q\}$ are distinguished. The modified Hopcroft approach may produce a somewhat smaller string than table-filling because it always goes for the smaller set when partitioning, whereas table-filling just does whatever it encounters first. However, a counterexample demonstrates that Hopcroft does not necessarily guarantee the absolute shortest witness: see Figure 17 and Figure 18. (Note that these must be converted to DFAs before running the algorithm.) The shortest witness is the string 12. However, when the algorithm is run, it may produce either 12 or 010 as the witness. As an enhancement, it would be feasible to sort the list $L(a)$ of items to be processed, such that the smallest, shortest possibilities are considered first. This would increase the likelihood of finding the absolute shortest string, since the algorithm stops as soon as the path to some witness has been found. Alternatively, a quicker approach to find the shortest string is suggested in the (nearly) linear algorithm described later.

With the algorithm as it stands, what are the bounds on string length? Obviously, the empty string is the shortest possible witness (length 0), if exactly one of the two automata happens to have an accepting initial state. An upper bound can be determined from the total number of possible configurations. Although either of the automata might repeat a state and symbol pair while the witness is being formed, we cannot arrive at a scenario where both automata are repeating a configuration together. In other words, if state p from one automaton and state q from the other appear together as a pair more than once, and we continue to run the transition function δ on them, then we are stuck in a loop, and a witness should already have been found. The specific symbol a which was used to distinguish p from q does not matter here, because we are

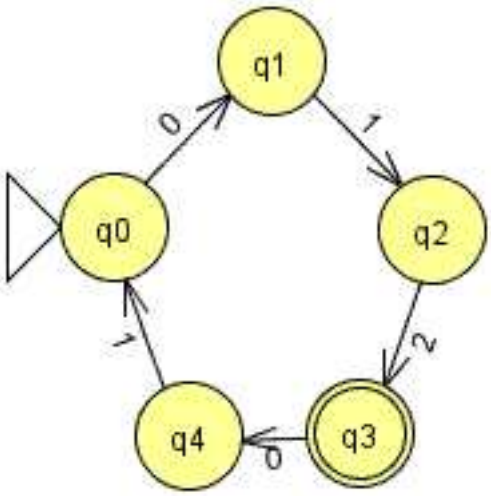

Figure 17: An NFA which accepts $012(01012)^*$.

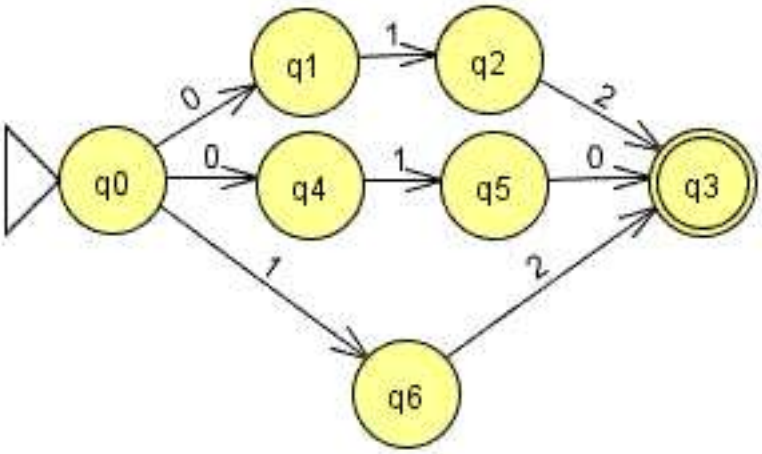

Figure 18: An NFA which accepts $012 + 010 + 12$.

only storing (at most) one symbol per pair of distinguishable states. Therefore, the maximum length of a witness produced via this algorithm is the size of the table used to store the symbols. If there are n_1 states in the first automaton, and n_2 in the second, this makes a maximum witness length of $(n_1 * n_2)$. (This applies to the table-filling algorithm as well, since we can disregard the symbols stored for state pairs where both states reside in the same DFA.) In our example comparing the DFAs which accept 00 and 01, each DFA has four states, so this makes a maximum length of $4 * 4 = 16$ symbols. The actual witness was only of length 2, so this is a loose upper bound. Observe that a tighter bound is the number of symbols actually stored in the table at program completion. If the partitioning algorithm is halted with only a handful of entries in the table, then that sets an upper limit on the quantity of possible symbols in the witness string.

Is there a useful way to quantify the maximum length based on the number of blocks in the partition? Whenever a pair of states is split into separate blocks, a symbol is entered into the table for that pair. Each block will contain one or more states. There appears to be no restriction saying that the witness could not be based on multiple symbols found from the same pair of blocks. In other words, the witness may include both symbols a_1 and a_2 where a_1 is in the *SYMBOLS* lookup table for states (p_1, q_1) and a_2 is the entry for (p_2, q_2), but both p_1 and p_2 are together in the same block and q_1 and q_2 are together in a different block. Since the DFAs are not necessarily minimal, there may be multiple equivalent states in the same block. Moreover, if the partitioning was halted early, it may be that p_1 and p_2 are not really equivalent, and/or that q_1 and q_2 are not equivalent. Yet they appear in the same block. Therefore, the number of blocks does not appear to produce a useful limit on the length of the string, as pairs of blocks may be repeated while the witness is created from pairs of states.

Finally, we need to account for the time to form the witness string (step 10). If the models M_1 and M_2 are equivalent, the time is zero. Otherwise, the time is no more than the length of the string, discussed above, where the transition function δ is executed twice for each character in the string. The longest possible string would be $n_1 * n_2$, which is much less than n^2 since $n = n_1 + n_2$. Adding that to the $O(n^2)$ from the previous steps, this still gives a maximum runtime on the order of $O(n^2)$ for the entire process.

A note on the implementation for this project: In JFLAP, recall that DFAs are not required to have a transition explicitly defined for every symbol for every state; missing transitions go to an implied dead (trap) state. There are two possible ways to handle this scenario. First, what happens if we just run the algorithm without the trap state? A problem arises when trying to construct a witness that may require the use of the dead state. For example, if a symbol a leads to a legitimate state in one automaton, but it is a dead end in the other automaton, then we have no idea what the next symbol in the witness should be, as the *SYMBOLS* table has not tracked any pairs of states which include the trap state. Therefore, we are forced to try the second alternative: If we add the trap state to the DFA before starting the algorithm, so that the

DFA is "complete," then it can be run as usual. (Observe that this is also the reason why NFAs must be converted to DFAs before the Hopcroft algorithm can be run. For each state, every alphabet symbol must lead to exactly one state to avoid ambiguity.) If we do not need a witness, then the trap state is not required. This is because, based on the transitions which are present, some path will still lead to a final state in one of the automata, but not in the other (assuming they are not equivalent), so the partitioning based on the inverse transition function still works. In the code which was written for this project, a new setUpTrapStates() method was created, rather than relying on the existing JFLAP 6.4 trap state code which was too deeply integrated into the GUI. A quick test verifies if the number of transitions equals the number of states times the size of the alphabet. If so, then all transitions are accounted for. Otherwise, the trap is actually needed. In that case, the complexity of this method is just $O(n)$; the program will loop through each state for each symbol (where the number of symbols is constant) to check what is missing and add it to the model if necessary. The amount of work for each addition is not entirely negligible, as the automaton, the function delta, and delta inverse must all be updated. However, these are single-step processes, and the overall complexity of the program remains $O(n^2)$.

3.3.7 An Observation on State Inequivalence

When the algorithms discussed above are used to test the equivalence of two DFAs, typically the focus has been on determining if the start states of the two automata are equivalent. Clearly, if the start states are equivalent, then both automata must accept the same language.

However, it is also possible to attack the problem from a different angle. Since the minimal DFA for a given language is unique, we can use any reachable state(s) as the criteria to judge equivalence, not just the start states. Observe that any states which are equivalent in a non-minimal DFA will be reduced to a single state to form the unique minimal DFA. There may be duplicates in a non-minimal DFA, and unreachable states must be eliminated, but each "minimized" state must be present at least once in the non-minimal version. Therefore, to test for equivalence of two DFAs (assuming unreachable states have already been eliminated), each state in the first automaton being compared must have at least one corresponding equivalent state in the second automaton, or they cannot recognize the same language.

This has a potential practical application if a programmer would like an extra shortcut to halt the partitioning algorithms sooner. For instance, in the Hopcroft $n \lg n$ algorithm, as soon as a new block B is formed, we can do a quick check, and if B contains only (reachable) states from a single DFA, then we know that the two DFAs are not equivalent, and we can stop executing the program. (It is possible that not all of the states within B are equivalent, but the important point is that none of them are equivalent to any states in the second DFA. However, unreachable states in either DFA should be ignored here, as they have no impact on which strings are accepted.)

Similarly, in the table-filling algorithms, it would also be possible to halt execution sooner in some cases, even before the start states are marked as distinguishable. If it is found that a particular (reachable) state has been marked distinguishable from every (reachable) state in the opposite DFA, then the two DFAs are not equivalent.

It would be interesting to run some empirical tests on this type of approach to see if it reduces runtime for different categories of DFAs. It does require a little more logic to be executed, but it may be worth it if it cuts down on the overall number of loops being performed. It may even be feasible to design a completely new equivalence-checking algorithm based on this property.

3.3.8 The (Nearly) Linear Algorithm by Hopcroft and Karp

Hopcroft and Karp demonstrate that, if it is not necessary to minimize the automata for some other purpose, it is possible to check for equivalence in (very nearly) linear time. This algorithm [12] does not appear to be as well known as the others, because it seems that more emphasis has generally been placed on minimization than on checking equivalence. This algorithm is quite different in that it employs a straightforward merging method instead of a partitioning approach. It begins by merging the start states into a single set (list), and it puts the pair of states on a stack for further consideration. Then, the stack is popped repeatedly until it is empty. For the current pair and for each symbol in the alphabet, the algorithm calculates δ to get the next pair of successor states. The successors are then merged and pushed onto the stack, if they have not already been merged. One state will be chosen as the representative of the merged states. At the end of the algorithm, if any merged set of states contains both an accepting state and a non-accepting state, then the two automata are not equivalent. (This resembles a proof by contradiction: we assume that the start states are not distinguishable, and put them into the same set. However, if this eventually results in a situation where accepting and non-accepting states are combined, then the start states must be distinguishable, since they cannot be equivalent.)

As an example, test Figures 8 and 9 for equivalence. The first step is to initialize eight sets, one for each state in the automata (q_a through q_h). Next, we merge the two start states, q_a and q_e, into a single set, represented by q_a, and push the pair $\{q_a, q_e\}$ on a stack. Then we enter a processing loop: pop items off of the stack and process them until the stack is empty. First, pop $\{q_a, q_e\}$. Follow each of the transitions out of these states and merge the sets containing the resulting pairs (if they are not already merged). New merges must always be added to the stack for further consideration. On symbol 0, the transitions from q_a and q_e lead to the pair q_b and q_f, so merge the new pair into a new set represented by q_b, and push them onto the stack. On symbol 1, the transitions lead to q_d and q_h; these must also be merged and pushed. Assume the set is represented by q_d.

For the next iteration of the loop, we pop $\{q_d, q_h\}$. On both symbols 0 and 1, these states only lead to themselves, and they have already been merged,

so there is nothing new to push onto the stack. Continuing, pop $\{q_b, q_f\}$. On symbol 0, they go to $\{q_c, q_h\}$, so the new pair is merged (into the same set with state q_d, which has already been combined with q_h), and $\{q_c, q_h\}$ is pushed onto the stack, assuming q_c is the new representative of the set. The representative member may vary, depending on the set-merging algorithm which is used. On symbol 1, the transitions lead to $\{q_d, q_g\}$; again, combine the states into a single set $\{q_c, q_d, q_g, q_h\}$. If q_c is still the representative (name) of the set, the pair to place on the stack is $\{q_c, q_g\}$.

The next pop is then $\{q_c, q_g\}$. For both input symbols 0 and 1, they transition to $\{q_d, q_h\}$, which are already in the same set. Therefore, pop the next pair, $\{q_c, q_h\}$. These states also always lead to $\{q_d, q_h\}$ on any input symbol. There is nothing more to place on the stack, and the stack is now empty. Our final sets are: $\{q_a, q_e\}$, $\{q_b, q_f\}$, and $\{q_c, q_d, q_g, q_h\}$. To test for equivalence, we scan through the sets and determine if any of them contain both accepting and non-accepting states. The third set does contain both, and therefore, the two automata are not equivalent.

Since all input symbols (i.e., all transitions) are considered for each pair of states on the stack, every reachable state will eventually be pushed onto the stack. The first time a new state is encountered, it is alone in its own set, so it is its own representative, and its actual transitions will all be followed. Unreachable states do not impact the language accepted by the DFA, so they are irrelevant when testing equivalence. Therefore, the algorithm correctly and completely determines whether or not the two DFAs accept the same set of strings.

Note that to be linear in complexity, this algorithm requires a linear set merging algorithm (or "list" merging algorithm). In [12], Hopcroft and Karp do not provide a set merging algorithm, but instead refer the reader to [13], a technical report by Hopcroft and Ullman. Hopcroft and Karp briefly explain the MERGE operation involved when combining sets and the FIND operation used for finding a particular element's set:

> "The linear list merging algorithm starts with n sets, each set consisting of a single integer between 1 and n. The set containing the integer i is given the name i. The list merging algorithm executes two types of instructions, a merge instruction and a find instruction. The execution of an instruction MERGE(i, j, k) causes the set named i and the set named j to be combined into a single set named k. The execution of an instruction FIND(i) determines the name of the set currently containing i. The important property of this algorithm is that the time necessary to execute any sequence of merge and find instructions, whose length does not exceed a constant times n, is bounded by a constant times n." [12]

However, as subsequent research has shown, no known algorithm can quite do this within a constant times n. Rather, the best complexity for this particular scenario would be $O(n\alpha(n))$, where $\alpha(n)$ is a function which is so slow-growing

that it is constant for all practical purposes. In fact, Hopcroft and Ullman published a later paper [14] on set merging algorithms which does not assert that the complexity is linear. The most up-to-date version, specifically disjoint-set forests as presented by Cormen et al., will be used here. Cormen et al. explain the tightest possible known bound on the complexity, and their pseudocode resembles [13] but avoids set naming concerns while remaining extremely clear and concise. They describe several approaches, and we will use the fastest version, with two heuristics that improve the run time ([5], p. 508). Read Cormen et al., chapter 21, for the full details on the run time. We will review the pseudocode in a moment. In brief, MAKE-SET(i) creates a new set with i as the sole member, and i is specified as its own parent. In this model, the root is the representative member (or name) of any given set. In place of MERGE(i, j, k), we will use UNION(i, j), which combines the trees containing i and j in an efficient manner; the name of the new set will be the root of one of the two original trees. Finally, instead of FIND(i), the function to find the representative for the set containing i will be called FIND-SET(i). The pseudocode for the equivalence checker is then as follows. n is the sum total of the number of states in the two automata which are being checked for equivalence.

1. On input (M_1, M_2), where M_1 and M_2 are DFAs with start states p_0 and q_0 respectively:
2. Initialize n sets: For every state q, MAKE-SET(q).
3. Begin with the start states. UNION(p_0, q_0) and push the pair $\{p_0, q_0\}$ on a stack.
4. Until the stack is empty:
   (a) Pop pair $\{q_1, q_2\}$ from the stack.
   (b) For each symbol a in Σ:
      i. Let r_1 = FIND-SET($\delta(q_1, a)$) and r_2 = FIND-SET($\delta(q_2, a)$), so that r_1 and r_2 are the names of the sets containing the successors to q_1 and q_2.
      ii. If $r_1 \neq r_2$, then UNION(r_1, r_2) and push the pair $\{r_1, r_2\}$ on the stack.
5. Scan through each set. If any set contains both an accepting state and a non-accepting state, then the two automata are not equivalent. Otherwise, the automata are equivalent.

Hopcroft and Karp initially argued that the complexity of this algorithm is linear, $O(\|\Sigma\|n)$, where $\|\Sigma\|$ is the size of the alphabet. It can actually be implemented in a manner such that it is virtually linear. Clearly, it only takes linear time on n to initialize the n sets. The initial merge of the start states will definitely take less than linear time, as only two sets of size 1 are involved. Also, the final scan to check for combinations of accepting and non-accepting

states in the same set will only have to consider n states at most, so it is linear in the worst case. The tricky part is the loop where items are being popped from the stack and processed. As explained above, the work for the FIND-SET and UNION calls is known to be bounded by the number of calls (i.e., no more than some multiple of n) times a slow-growing function on n which is very close to constant. The number of pushes to the stack will be no more than $n - 1$ because we start out with n sets, and each time a new pair of states is pushed, the number of sets has been reduced by one. When $r_1 = r_2$, nothing will be pushed on the stack. Therefore, no more than $n - 1$ pairs will ever be pushed (popped), and the total number of calls for this outer loop is $O(n)$. This is multiplied times $\|\Sigma\|$ for the inner loop, giving a total complexity which closely approximates $O(\|\Sigma\|n)$, or nearly $O(n)$ if we leave out the constant $\|\Sigma\|$.

This merging algorithm appears to be the fastest one available for testing DFA equivalence. (It would be difficult for it to be much faster, since it takes a minimum of linear time just to read through the n states!) The pseudocode even has the advantage of being elegant and easy to understand. One possible programming shortcut would be to stop execution as soon as an accepting state was merged with a non-accepting state.

What does the set merging code look like? Cormen et al. provide the following pseudocode. The set elements x and y are assumed to be integers, for the sake of simplicity. p is an array storing the elements' parents, and *rank* is an array setting an upper bound on the distance that a set's root element might have from the farthest leaf of its tree. Whenever two sets are merged, the *rank* allows optimization via a "union by rank" heuristic wherein the root of the larger set remains the root of the combined set. The other important heuristic is "path compression;" whenever a FIND-SET call is made to find an element, the root becomes the direct parent of each node which is recursively encountered along the way. Here is the pseudocode exactly as presented in Cormen et al. (quoted directly from [5], p. 508):

MAKE-SET(x)

1. $p[x] \leftarrow x$
2. $rank[x] \leftarrow 0$

UNION(x, y)

1. LINK(FIND-SET(x),FIND-SET(y))

LINK(x, y)

1. if $rank[x] > rank[y]$
2. then $p[y] \leftarrow x$
3. else $p[x] \leftarrow y$
4. if $rank[x] = rank[y]$
5. then $rank[y] \leftarrow rank[y] + 1$

FIND-SET(x)

1. if $x \neq p[x]$
2. then $p[x] \leftarrow$ FIND-SET($p[x]$)
3. return $p[x]$

3.3.9 A (Nearly) Linear Algorithm, with Witness

With a little work, we find that the previous algorithm can be modified to produce a witness string, without slowing it down significantly or making it overly cumbersome. Observe that the basic algorithm begins by processing the start states, and then it follows transition paths (strings) as they extend to all reachable states. If we can track a string that leads from the start states to an accepting state in one DFA, but to a non-accepting state in the other, then we will have a witness.

Notice that there is no reason why the list of items to be processed must specifically be stored on a stack, rather than some other data structure. The important thing is to merge the states as we encounter them, not to track the order in which the states are encountered. Therefore, the first modification would be to replace the stack with a FIFO queue. This will ensure that strings of length 1 are processed before strings of length 2, which are processed before strings of length 3, etc., so that we can find the shortest possible witness. Essentially, we are now doing a breadth-first search to try to find a witness. (Note that this modification is not necessary if we do not wish to restrict the witness length.) Second, the actual pair of states p and q should always be placed on the queue, rather than r_1 and r_2, which are just are the names of the sets containing p and q. This is important for our purposes, since we need to strictly and systematically follow both of the DFAs' transition functions in subsequent iterations in order to produce an accurate witness. To emphasize this point, the pairs will now be ordered ((p, q) rather than $\{p, q\}$): p always comes from one DFA, and q from the other. Third, the merge procedure remains the same. However, execution should stop as soon as we merge a pair (p, q) if exactly one of the two states p and q is accepting. That is our flag that the two automata are inequivalent and that we have found a witness. Fourth and finally, we need a way to store the transitions which were followed, so that the witness can be reconstructed at the end of the program. A structure such as a doubly linked tree might work, with state pairs as the nodes and input symbols as the links, but the pseudocode below assumes that a map (such as a Java HashMap) is being implemented.

1. On input (M_1, M_2), where M_1 and M_2 are DFAs with start states p_0 and q_0 respectively:
2. Initialize n sets: For every state q, MAKE-SET(q).
3. Begin with the start states.

(a) If p_0 and q_0 are both accepting states, or both non-accepting states, UNION(p_0, q_0) and place the pair (p_0, q_0) on a FIFO queue.

(b) Otherwise, output the empty string as the witness. Halt.

4. Until the queue is empty:

   (a) Dequeue pair $\{q_1, q_2\}$.

   (b) For each symbol a in Σ:

      i. Let $p = \delta(q_1, a)$ and $q = \delta(q_2, a)$.

      ii. Let $r_1 =$ FIND-SET(p) and $r_2 =$ FIND-SET(q), so that r_1 and r_2 are the names of the sets containing the successors to q_1 and q_2.

      iii. If $r_1 \neq r_2$:

         A. UNION(r_1, r_2) and place the pair (p, q) on the end of the queue. In a map, save a key-value pair with (p, q) as the key, and (q_1, q_2, a) as the value. The value can later be used to reconstruct how the key pair was reached.

         B. If p and q are both accepting states or both non-accepting states, continue looping.

         C. Otherwise, p and q are distinguishable, so create the witness string as follows:

         D. Initialize the witness to the empty string.

         E. Initialize KEY $= (p, q)$.

         F. Until KEY = the start states (p_0, q_0): look up the map value (q_{K1}, q_{K2}, a_K) corresponding to KEY. Concatenate the symbol a_K to the *beginning* of the witness. Set KEY $= (q_{K1}, q_{K2})$, and repeat.

         G. Output the witness and halt.

5. The automata are equivalent. Output "equivalent, no witness" and halt.

Let us rework the example of Figures 8 and 9. We initialize one set for each state, $\{q_a\}$ through $\{q_h\}$. Then, merge the start states, q_a and q_e, into a single set. Place the pair (q_a, q_e) on a queue. Now we loop until the queue is empty. Take (q_a, q_e) from the queue, and follow the transitions on each input symbol. Symbol 0 takes us to q_b and q_f. These need to be merged, and the pair (q_b, q_f) is placed on the queue. In addition, put the key-value pair $((q_b, q_f), (q_a, q_e, 0))$ in a map. Both q_b and q_f are non-accepting states, so we keep going. On input 1, (q_a, q_e) transition to q_d and q_h. These two state sets are merged, and (q_d, q_h) is added to the queue. $((q_d, q_h), (q_a, q_e, 1))$ is added to the map. Since q_d and q_h are both non-accepting states, we continue with the loop. We dequeue the next item, (q_b, q_f). On input symbol 0, these states transition to q_c and q_h, so we merge these states into the set $\{q_c, q_d, q_h\}$, and put (q_c, q_h) on the queue. Then we add $((q_c, q_h), (q_b, q_f, 0))$ to the map. Finally, we find that q_c is an accepting

state, but q_h is not, so these states cannot be equivalent, and we can produce a witness.

To do this, we initialize the witness to the empty string, and begin with the last pair, (q_c, q_h). If we look this up in the map, we get the value $(q_b, q_f, 0)$, meaning the pair was reached on symbol 0, coming from (q_b, q_f). Add the symbol 0 to the beginning of the witness, and then find (q_b, q_f) in the map. This key gets us the value $(q_a, q_e, 0)$. So, we add another 0 to the beginning of the witness, giving the string 00. Since (q_a, q_e) is the pair of start states, we have returned to our origin, and the final witness string is 00. This string is accepted by exactly one of the two DFAs.

The complexity of this new algorithm is still near-linear. It should only take constant time to verify if a particular state is accepting. Assuming map additions and lookups can be done in near-constant time, the overall complexity will remain nearly linear. The loop to create the witness will only be executed $O(n)$ times – no more than the length of the witness itself, which can be no longer than the number of items placed on the queue in the main loop. Again, the queue implementation ensures that we find one of the shortest possible witnesses by checking the shortest strings first. Space-wise, the big addition is the map, which is also bounded in size by the number of items placed on the queue.

In the EquivalenceMergeWitness.java class written for this project, the queue was implemented as a Vector, and a counter was used to track the current head of the queue. To ensure that a dequeue operation was a constant time operation, dequeuing was performed by getting the element at the index of the counter and then incrementing the counter, rather than by physically removing elements and reindexing/resizing the Vector every time states were dequeued. Since JFLAP does not store transitions as a transition function, a HashMap *delta* was built to map an input state and input symbol to the output state; this is the same approach as in the EquivalenceNlgNWitness.java program. The merge algorithm, however, required no inverse transition function, and it was much easier to implement. The number of lines of code in the merging program was almost cut in half by comparison to the Hopcroft partitioning algorithm. This includes comments, but leaves out supporting objects such as the disjoint set class, which was a separate file.

In the actual set implementation, StateDisjointSets.java, Vectors were used for the parents and the ranks, rather than arrays. This was done so that the number of elements did not have to be known immediately when initializing the data structures. Unlike arrays, Vectors can grow. This helped to avoid extra iterations to process the JFLAP States and Transitions, especially in cases where a trap state had to be added later on. Moreover, since the set elements were States rather than integers, an additional HashMap *stateToIndex* was used to quickly map each State to its Vector index. (The Java HashMap is actually a hash table, and it was a good choice because it "provides constant-time performance for the basic operations (*get* and *put*), assuming the hash function disperses the elements properly among the buckets." [10]) To facilitate reverse lookups, a Vector of States called *indexToState* was implemented. The

position of each State element in this Vector matched its index in the *parent* and *rank* Vectors.

3.3.10 Testing Equivalence in JFLAP

The original JFLAP program already contains code which can be used to check finite automata to determine if they are equivalent. Here is a brief overview and discussion of the most significant of these Java classes.

The DFAEqualityChecker is a simple isomorphism checker that verifies if two automata are identical or not, apart from the naming of states. It cannot be used to check for equivalence unless the automata have already been converted to (or created as) minimal DFAs, since the NFAs to specify a given language are not unique in general, and only the minimal DFA is a unique DFA. Like much of the code in JFLAP, this DFAEqualityChecker class treats the automaton as a directed graph, where the states are the vertices (nodes) and the transitions are the edges. It compares pairs of states recursively, beginning with the initial states of the two DFAs, to see if they correspond. The *hypothesize*() function operates on these states as follows:

1. It checks a hashmap to see if the pair of states is not already known to match (or not match). If found, it returns true (or false).

2. Otherwise, if one of the pair is accepting and the other is not, then it returns false, as they cannot be equivalent.

3. Otherwise, if the number of transitions exiting each state in the pair is not equal, then it returns false. (Recall that JFLAP allows missing arrows in DFAs; they lead to an implied dead state. Therefore, for two states to be equivalent, they need to have the same quantity of exiting transitions.)

4. Otherwise, the transitions must be tested in more detail to verify equivalence. If each transition from the first state does not have a corresponding transition from the second state, then it returns false, as the states would not accept the same strings. To decide this, the program first checks the labels on all of the transitions. If there is a transition on some input a from the first state but no such transition from the second state, then false is returned. Otherwise, the two states are placed in the hashmap as potential matches, and a recursive call is made to this *hypothesize*() function to verify if the new state reached in the first DFA on each input a corresponds to the new state reached in the second DFA. If not, it removes the original pair of states from the map and returns false.

5. Otherwise, it returns true.

The JFLAP FSAEqualityChecker class makes use of this DFAEqualityChecker to verify whether or not two automata actually accept the same language. As you would expect, it converts the two automata to DFAs, minimizes each of them, and then runs the DFAEqualityChecker. Since the minimal DFA

is unique for a given language, this isomorphism-checking procedure will return the correct answer (equivalent, or not equivalent). This runs rapidly enough to work for typical school assignments containing only a handful of states, although, as we have seen, faster methods are available to handle large automata.

The NFAToDFA class is an implementation of NFA-to-DFA conversion. The algorithm is a subset construction approach along the same lines as the one described earlier in this chapter.

The Minimizer class, however, views partitioning in a slightly different way than the other partitioning algorithms in this paper. Rather than filling in a table or constructing linked lists, it uses a tree model. The root starts out containing all states, and the tree branches out whenever states can be distinguished on some input. The leaves of the finished tree represent groups of indistinguishable states, which can then be used to construct the minimized DFA. The JFLAP manual [28] by Rodger and Finley outlines this approach as follows (quoted directly from pp. 28 - 29):

> We describe the algorithm to convert a DFA M to a minimal state DFA M'.
>
> 1. Create the tree of distinguished states as follows:
>   (a) The root of the tree contains all states from M.
>   (b) If there are both final and nonfinal states in M, create two children of the root - one containing all the *nonfinal* states from M and one containing all the *final* states from M.
>   (c) For each leaf node N and terminal [symbol] x, do the following until no node can be split:
>   i. If states in N on x go to states in k different leaf nodes, $k > 1$, then create k children for node N and spread the states from N into the k nodes in indistinguishable groups.
> 2. Create the new DFA as follows:
>   (a) Each leaf node in the tree represents a state in the DFA M' with a label equal to the states from M in the node. The start state in M' is the state that contains the start state from M in its label. A state in M' is a final state if it contains a final state from M in its label.
>   (b) For each arc [arrow] in M from states p to q, add an arc in M' from the state that has p in its label to the state that has q in its label. Do not add any duplicate arcs.

The tree can be presented in a visual way which many students may find helpful. However, notice that the tree-building portion of the algorithm is fundamentally similar to the table-filling algorithm, although it looks very different. Here are some similarities:

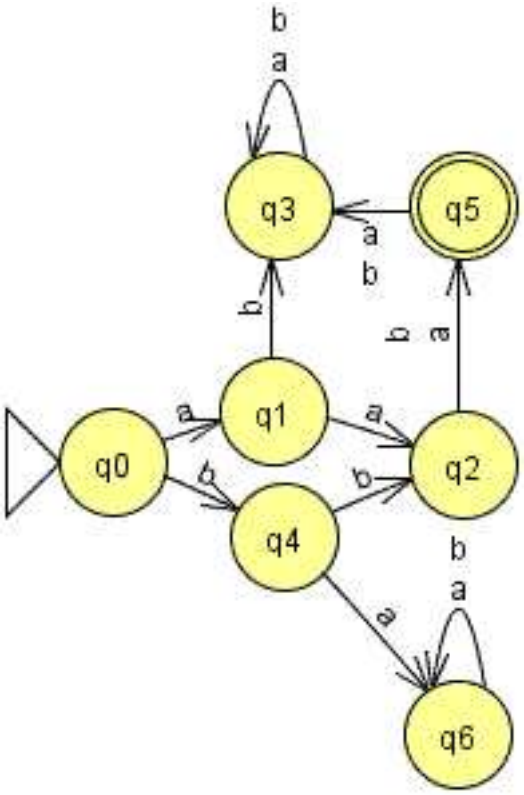

Figure 19: A nonminimal DFA that accepts *aaa*, *aab*, *bba*, and *bbb*.

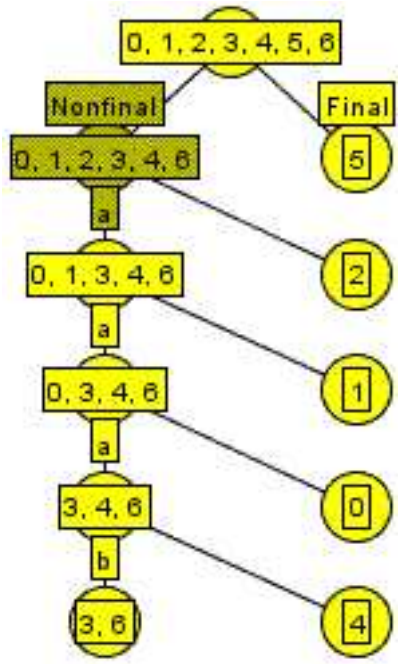

Figure 20: The JFLAP minimization tree for the DFA that accepts *aaa*, *aab*, *bba*, and *bbb*.

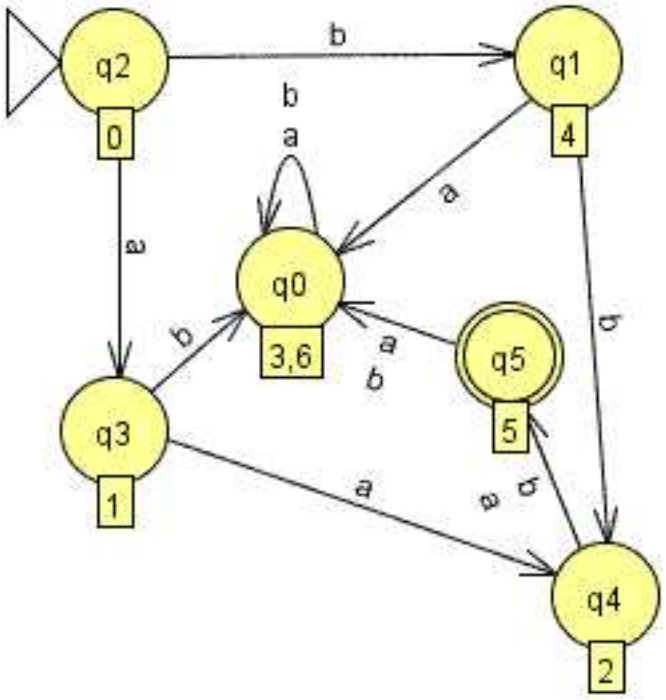

Figure 21: The minimized DFA produced by JFLAP.

- JFLAP simply splits distinguishable states into separate nodes rather than marking them distinguishable by filling in an 'X' in a table. At the start of the JFLAP program, the two immediate children of the root correspond to the initialization step in the table-filling algorithm where table entries are marked if one state is distinguishable and the other is not.

- As either algorithm runs, it repeatedly loops through every pair or set of states which has still not been distinguished, using every input symbol, until no more states can be distinguished.

- The "partitioning" is more visible in JFLAP, but recall that table-filling is also a partitioning algorithm because it ultimately divides the states into unmarked groups which have been determined to contain only equivalent states.

- One difference is that JFLAP doesn't just look at pairs of states. It notices when three (or more) states in a group (node) lead to two, three, or more different nodes, and separates them accordingly. The table-filling approach just marks anything distinguishable with a generic 'X.'

- In the table-filling algorithm, indistinguishable states are never marked. In this JFLAP algorithm, indistinguishable states remain in the same node in the finished tree. In either case, of course, the indistinguishable states may be represented by a single state in the minimized DFA.

When creating the new DFA, the "label" in the JFLAP algorithm is simply a list of the states contained in the node. For example, see Figure 19. This DFA accepts strings of length three beginning with either aa or bb. States q_3 and q_6 are the only equivalent states, so they are in the same leaf of the finished tree in Figure 20. The minimized DFA produced by JFLAP is shown in Figure 21. There are six states corresponding to the six leaves of the tree, labeled with the numbers of the states from the original DFA.

What is the time complexity of this algorithm? The pseudocode from the manual is very high-level, so the complexity could vary. Since it was not implemented with an eye to efficiency, JFLAP does not come close to the $O(n^2)$ or $O(n^4)$ which is feasible with table-filling. Looking at the actual Minimizer.java code in detail, it appears to do extra initialization steps which are essential, but not obvious. An AutomatonChecker is run to check for NFAs. It counts the quantity of nondeterministic states by looping through each state and checking if any of its transitions are lambda (epsilon) transitions, or if any of the transitions occur on the same alphabet symbol. If there are any nondeterministic states, the program aborts. Otherwise, once it decides that it has a DFA, not an NFA: It first finds unreachable states via the UnreachableStatesDetector (more on that algorithm in a moment) and removes them. Since the JFLAP GUI allows multiple symbols on a single transition, the program must run a FSALabelHandler to replace these with separate transitions (and potentially more states). This is not what one would expect from the standard definition

of DFAs, so we could disregard this step in our analysis. However, since the JFLAP GUI (and some standard definitions) also allow missing transitions for a DFA, the code then checks if a trap state is needed, an $O(n)$ operation whose complexity is affected by the number of input symbols and the quantity of transitions, which will be no more than a constant. If it turns out that transitions are missing, then a trap state is added in another $O(n)$ operation (a loop for every state for every symbol).

Now that it is a "minimizable automaton," the algorithm begins in earnest. The user has the convenient option of building the tree in sections or all at once. The first step grabs an array containing all of the states and puts it in the root. This would take a single unit of time, except that for some reason the Arrays.sort() method is used to put them in ascending order by state ID. The first-level child nodes are an array containing the final states (pulled directly from an existing HashSet) and an array of the non-final states (which is determined via an $O(n)$ loop that checks the HashSet of final states against each state in the DFA). These children are inserted into the tree. Insertions appear to be an efficient operation as the Swing DefaultTreeModel is used. Now a while() loop begins, and it runs as long as the tree is not minimized yet, meaning some group of states in a node is still distinguishable. Unfortunately, each call is run identically twice, once to get a boolean isMinimized, and again to obtain the actual group of states. This is expensive because it requires getting all of the leaves (a recursive call starting at the root), iterating through all of the leaves, finding the alphabet (by iterating through all of the transitions in the DFA each time!), and testing each group (leaf) against each alphabet symbol for splittability. Splittability is detected by looping through every transition of every state in the group to see if it has the desired alphabet symbol. If it does, JFLAP will track the node it goes to, and put this node on a list. This again causes extra work, as it will get all of the leaves (each time) and iterate through them until the correct node for the resulting state is found. If more than one node exists on the final list, the group can be split.

Unfortunately, the program does not save the alphabet symbol that was used, so it has to look for one all over again. It retrieves the alphabet from the automaton from scratch again, and rechecks the group for splittability on each symbol all over again. Once it has the symbol, it calls much of the same kind of messy logic to get and iterate through the leaves again and determine which states in the group lead to different leaves on this symbol. These are then added to the tree as separate children of the group, with a notation of what the alphabet symbol was. This is not quite as simple as it could be, as it does another recursive call through the tree to find the node for that group which we are using as a new parent! In short, the complexity of this code is some polynomial on n – a polynomial far greater than it truly should be. It does not pose a problem for the typical JFLAP user because the automata are so small. There is no doubt that the same pseudocode to create the tree could be implemented more efficiently to allow fast processing for larger DFAs, even if the user were still given the ability to build the tree piecemeal, so that they could watch the steps.

Now that the tree has been formed, the minimal DFA can be created. JFLAP searches for all of the leaves in the tree (again), and iterates through them. The program loops through the states within the leaf to determine if it contains an automatically created trap state. If so, this leaf is omitted. Otherwise, it creates a state in the minimal DFA for that leaf. JFLAP loops through the states within the leaf again to see if any of them are initial, and if so, the new DFA state becomes initial. Similarly, it also loops through the states in the leaf to determine if there are any final states; if so, the new state becomes final. These three loops could of course be combined into one for somewhat tighter efficiency, but clearly there are other tasks which would have a significantly more dramatic effect on performance if their output was saved to minimize repetition.

To find the transitions for the minimal DFA, JFLAP creates an empty list for the transitions and then loops through the new set of states. For each state, it gets the transitions by looking up the tree group for that state, and then picking the first (original) state within that group. It gets the array of transitions for that state from a map. (It seems that one original state suffices, since the states are all equivalent within a group, and their sets of transitions must therefore be equivalent, in effect. This is, in fact, a time-saver.) For each transition, it determines what old state it went to, and which new state group now is the destination (it contains the old state). If the group does not contain a temporary trap state, it finds the minimal DFA state for that group, determines the alphabet symbol for the transition, and adds the transitions to the list. The program returns to the top of the loop to the next new state, if any. Lastly, the program iterates through the full list of new transitions and adds them to the minimal DFA, one by one, and the DFA is done. Things could be tightened up, but we are still looking at some polynomial on n to create the minimal DFA from the tree. The formation of the tree itself appears to be the most costly part of the implementation, both because of the inherent complexity of the steps and because of the extreme duplication of calls.

An additional point to make about the JFLAP equivalence testing algorithm: since JFLAP allows missing arrows in DFAs, some special handling is required. Any missing arrows must be directed to a temporary dead ("trap") state before the tree is built during the minimization process. States q_3 and q_6 in Figure 19 were both dead states. If the user runs the "Minimize DFA" menu option of the software, and they have any missing transitions in their automaton, they will actually see that an extra trap state is added to their automaton until the tree is done. However, this trap state is automatically removed before the finalized minimal DFA is displayed. Therefore, the minimal JFLAP DFA for the same language can vary. For example, see Figures 22 and 23, which can both represent the languages with alphabet $\{0, 1\}$ consisting of an even (nonzero) number of zeroes. In this case, the symbol 1 does not even appear in the second model. The strict definition of the DFA requires transitions to appear for each symbol for each state. JFLAP, however, considers both of these models to be valid. Using the JFLAP algorithms, the minimal DFA will be one state smaller if the user originally entered a diagram with some dead-end arrows missing. This happens because if a leaf in the tree contains a

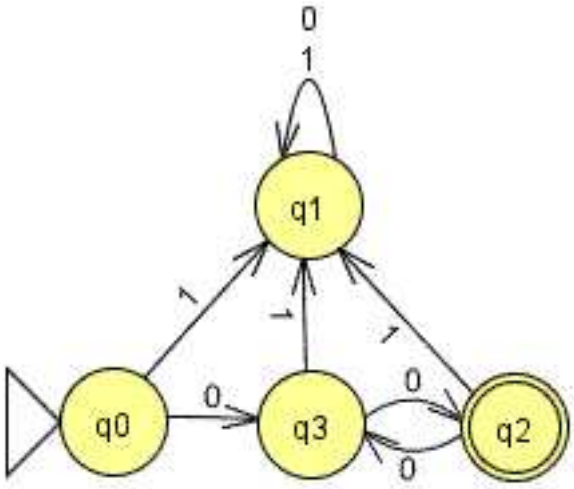

Figure 22: A DFA that accepts an even number of zeroes. q_1 is a trap state.

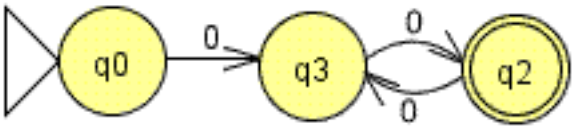

Figure 23: Another DFA that accepts an even number of zeroes. Legal in JFLAP even if 1 is in the alphabet.

temporary trap state, no corresponding state is added to the new minimal DFA, and none of the transition arrows to this state are drawn. On the other hand, if the user enters a DFA with all transitions accounted for, and there is a dead state involved, this state will NOT be eliminated in the final diagram, as the program does not consider it a special "trap." Yet the "Compare Equivalence" menu option will recognize that the DFAs represent the same language, whether or not the dead state is shown. However, the two DFAs are certainly not isomorphic. Since the overall method is to compare graph isomorphism, and the third step of DFAEqualityChecker.hypothesize() relies upon a count of the number of transitions (edges) to compare pairs of states (nodes), JFLAP first has to define each minimal DFA to be in the form which contains no arrows which would lead to a dead state. Then, behind the scenes, the two DFAs are correctly tested against each other for equivalence.

Stepping back from the specifics of the Java implementation for a moment, at a higher level, it is easy to modify and speed up the JFLAP pseudocode for minimization if all we want to do is check for equivalence of DFAs. If we took the Minimizer algorithm and inputted the two DFAs to compare, we would just need to build the tree and then check if the two start states appear in the same node or not. Again, this is simply the table-filling algorithm in the guise of a tree. There is no need to check for isomorphism (or to worry about trap states). Here is the modified pseudocode:

1. On input (M_1, M_2), where M_1 and M_2 are DFAs:
2. Create a tree which will branch on distinguishable states. The root node of the tree contains all of the states from M_1 and M_2.
3. If there are both final and nonfinal states in the root node, create two children of the root - one containing all of the *nonfinal* states and one containing all of the *final* states.
4. For each leaf node N and symbol x, do the following until no node can be split:
   (a) If states in N on x go to states in k different leaf nodes, $k > 1$, then create k children for node N and spread the states from N into the k nodes in indistinguishable groups.
5. If the start states of the two DFAs are in the same leaf, then M_1 and M_2 are equivalent. Otherwise, they are not equivalent.

The two automata accept the same set of strings (the same language) if and only if the start states are equivalent, so we are done. The bulk of the work for this revised algorithm takes place in step 4. If the code were written carefully, so that transitions could be determined quickly and nodes could be found/inserted quickly, this would probably approach the n^4 or n^2 table-filling algorithms in terms of efficiency. As we have seen, it would depend on the implementation of

the algorithm, including the representation of the objects (states, transitions, symbols, leaves) and how we do lookups.

Back to minimization: The UnreachableStatesDetector is another Java class of interest when minimizing in JFLAP. It can be used to determine which states are unreachable from the start state. Since these states will never be used to process any input string, they can safely be deleted from an automaton as a first step when minimizing. JFLAP uses another graph-oriented approach to do this. When running the UnreachableStatesDetector program on an automaton, all states in the automaton are initially unmarked (colored white). Then it uses a depth-first search, beginning at the start state, to recursively visit and mark (color black) all neighbor states that can be reached by following all specified transitions. Whenever a new state is reached, its neighbors are also visited and marked. At the end of the recursion, any state that is still not marked (white) is unreachable, and may be deleted without changing the language which the automaton represents.

3.3.11 Batch Grading: Testing Equivalence in BFLAP

To facilitate grading of student assignments done in JFLAP, Ambrose Kofi Laing wrote a Perl program for batch grading [21], called BFLAP. It includes code to check for equivalence of DFAs by taking the symmetric difference and then testing for emptiness, as described in the first equivalence-checking algorithm above. However, if this is the only algorithm the teacher uses, they may be faced with a dilemma when the student's answer does not match the answer key. Should the student receive any credit for their attempt, or should they simply receive a zero?

BFLAP has a way to give partial credit. It simply runs several strings on the student's automaton to see if the outcome (accept/reject) matches the expected outcome. The student receives points for each string which is correctly accepted or rejected. As Laing comments in the "Quantitative Grading Section" of his Perl script [21], "everybody gets points taken off for exactly the same reasons, regardless of how their machine was designed. I test your machine on a number of strings (choose the same strings for everyone, and I make up the set of strings before looking at the automata)." This is a good way to enforce uniform, fair grading. However, Laing adds that "You can get a perfect score on this section with luck, even if your machine is not correct." So, a potential problem arises if this is the only algorithm used during the grading process; the student could submit an incorrect answer, and never realize that it was wrong if all of the selected strings happen to pass the test. Clearly, this would not be ideal for learning purposes. Therefore, running the equivalence-checking algorithm first is important. Equivalence should be proved, not assumed on the basis of a few strings which may be only a tiny subset of the language (or its complement). Moreover, the instructor may wish to choose the test strings with care to reveal specific characteristics of the desired DFA.

3.4 Space

As we have seen, the equivalence (or inequivalence) of two DFAs can be determined in polynomial, $n \lg n$, or even near-linear time on the total number of states in the two DFAs, n. This implies that it can also be determined within almost linear space on the length of the input (number of states). Conceptually, in a Turing machine, you can only read and write to at most one tape cell per unit time. Therefore, it is not possible to use more space than time as the algorithm runs. However, we do not even need linear space. Cho and Huynh [4] mention that the inequivalence problem for DFAs has been shown to be NL-complete. This means that the problem falls into class NL (also called NLOGSPACE), which includes the problems that can be solved by a nondeterministic Turing machine using only a logarithmic amount of space based on the length of the input. Moreover, "complete" means that all of the other problems in NL can be reduced to this problem, without exceeding the space limitation. The proof was done by Jones, Lien, and Laaser in 1976 [18], although they use a more general model than DFAs, the deterministic generalized sequential machine with final states, which includes an output function and an output alphabet.

NFAs are a more complicated situation than DFAs. They require much more time and space. For example, recall that the subset construction technique – which may be used to convert an NFA to a DFA as a first step in some of the equivalence algorithms – requires exponential time in the worst case. In fact, according to Garey and Johnson ([8], p. 265), the problem of "FINITE STATE AUTOMATON INEQUIVALENCE," specifically whether two NFAs A_1 and A_2 "recognize different languages," is not even known to be in NP. NP is the class of problems which have polynomial time verifiers and can be decided in nondeterministic polynomial time. Garey and Johnson indicate that the problem is, however, known to be NP-hard, meaning that all problems in NP can be reduced to this problem using some polynomial time function. In terms of space, they state that testing the inequivalence of NFAs has been shown to be PSPACE-complete. PSPACE is commonly defined as the class of languages that are decidable in polynomial space on the length of the input. This can be using either a deterministic or nondeterministic Turing machine; they both fall into the same class, since PSPACE = NPSPACE. Note that NP is a subset of PSPACE. For a problem to be PSPACE-complete, there are two requirements. First, it must be in PSPACE. Second, all other problems in PSPACE must be polynomial time reducible to it.

Garey and Johnson also indicate that the general problem of "REGULAR EXPRESSION INEQUIVALENCE," or "Do [regular expressions] E_1 and E_2 represent different regular languages?" has been shown to be NP-hard and PSPACE-complete (p. 267). This makes sense since NFAs can easily be converted to regular expressions in polynomial time, and vice versa. For details, see a textbook such as Sipser, pp. 66 - 76. JFLAP supports these kinds of conversions, and it can walk the student through them, step by step.

The original PSPACE-completeness proof is attributed to Kleene [19], before PSPACE terminology was actually in use. Stockmeyer and Meyer also presented

several complexity results related to finite automata; for instance, see [30] and [31].

4 JFLAP and other Finite Automata Tools

A wide variety of different software tools is available to demonstrate and manipulate finite automata.

JFLAP, the Java Formal Language and Automata Package, is the tool which was selected for this project. JFLAP is a very popular Java 1.4 / 1.5 Swing GUI from Duke University which allows the user to easily visualize and simulate several different kinds of models, including finite automata, Mealy and Moore machines, pushdown automata, Turing machines, and their corresponding languages. JFLAP documentation is readily available; a book [28] and an online tutorial [17] have both been published. Recently, in October, 2007, JFLAP won recognition as a finalist in the NEEDS Premier Award for Excellence in Engineering Education Courseware competition, and its developers have continued to enhance JFLAP functionality since then. Version 6.2 from January, 2008 provides multiple layouts for finite automata. Version 6.4 from July, 2008 upgraded to Java 1.5 and incorporated new features such as an "add trap state to DFA" option. This project extends the JFLAP GUI so that students can get immediate feedback as they enter their models into the system. They can verify that their NFA matches the language of an NFA defined in another file (presumably an answer key file supplied by the instructor) and receive a witness string if it does not.

Since JFLAP stores DFA/NFA models in simple XML files with a .jff extension, it was readily feasible to incorporate the grading program into the existing RIT "try" submission/grading software [26], so that teachers can create tests and homework assignments. The .jff files can be sent in to "try," or if the instructor prefers, they can just use the stand-alone Java grading program by itself. Instructions are available in the User's Guide. The approach was to use existing JFLAP code to read in and parse the files. The XML contains descriptions of the states and transitions, and these can be instantiated in Java in order to determine whether or not two finite automata are equivalent. Note that the .jff files require tags specifying graphical X-Y coordinates, but the grading program will essentially ignore the coordinates as it has no need to display the automata.

Several other nice tools had been briefly reviewed before selecting JFLAP. For example:

- One program is the Visual Automata Simulator program by Jean Bovet [3]. Since it is simpler and more user-friendly than JFLAP, it has a smaller learning curve for the student, but it also has fewer features. Its appealing interface allows the user to easily create and run DFAs, NFAs, and TMs. Although this program may be preferable to JFLAP for a very quick introduction to these models, it may not be as useful as a tool for

an entire quarter-long theory of computing course. Moreover, the XML files which are used to store the models are far more complex than those for JFLAP, and they contain references to program-specific code (Java objects). Therefore, any program to grade student submissions from the Visual Automata Simulator would be permanently tied to the original source code, and it may be more complex to write than a grader for JFLAP.

- The Finite State Automata Utilities Toolbox [33] was written in Prolog, but Java has the advantage of being cross-platform.

- The Finite State Automaton Applet [7] is very basic; although it allows the user to step through a string, it does not offer features such as conversion between NFAs and DFAs. It also does not save the models to an export file, so automated grading would not be very feasible or convenient.

- Prof. Toida and other contributors at Old Dominion University have created several useful interactive Web exercises [32] to help students learn about FAs and other concepts. For example, there is a convenient regular expression equivalence checker, and there are several questions regarding DFAs and NFAs, including conversions. The student can request feedback immediately on their responses. However, these tools are not designed for the creation and grading of exams, since there is a fixed sequence of specific questions which has been published to the Web, and student input is not stored as they go through the lesson. In addition, it is awkward for the student to type in information using the notation for the state transition table. The JFLAP GUI is a more intuitive, quicker way to enter FAs.

- Graduate students and teachers have previously undertaken a large number of other projects to electronically demonstrate and manipulate finite automata and other models. For one example of thesis work, see the Java Computability Toolkit, or JCT [27].

- Moreover, an article about a Language Emulator toolkit even presents a "Comparative Table" of seven instructional tools ([34], p. 137), most of which are not mentioned above.

Note that these programs are not generally designed to allow submission and grading of problems, with the possible exception of FAdo, which uses a naive algorithm, discussed in "Equivalence to Grade Assignments" above. JFLAP appears to be one of the most recognized and established programs, so it is a good choice for this project. It does have an equivalence checker already, but its performance could be improved by replacing the existing Java classes with a faster algorithm or a faster implementation. Moreover, the grading utility may be useful at RIT, and perhaps at other universities where JFLAP is part of the instructional toolbox.

5 Conclusions

In this project, several different equivalence-testing algorithms for DFAs and NFAs were discussed. The time complexity to check two DFAs for equivalence ranged from a polynomial on n, such as $O(n^4)$, to essentially linear on n, where n is the total number of states in the two automata. It was observed that the time complexity (chiefly the exponent on the polynomial on n) could be significantly impacted not only by the underlying algorithm, but also by the details of how the algorithm was physically implemented. To reduce runtime, unnecessary calls should always be avoided.

In order to provide detailed feedback to students, a modification to the table-filling algorithms was suggested so that a witness string could be produced in cases where two finite automata were found to be inequivalent (for example, if the student had an incorrect DFA which did not match the answer key). A similar modification was then proposed for the faster $O(n \lg n)$ Hopcroft algorithm and for the near-linear algorithm; these two revised algorithms were physically implemented in Java to work with JFLAP .jff files, which represent models such as finite automata. Scripts and detailed instructions were provided in the User's Guide to allow instructors to use the new grading tool with the "try" submission/grading system at RIT. A menu option was also added to the JFLAP program to obtain a witness directly within the GUI. Moreover, the program was designed so that it could be run standalone from the command line, if so desired. In verbose mode, the user is able to trace the operation of the two different equivalence algorithms and see exactly how the witness strings are formed, or find out why the two automata being compared are truly equivalent.

6 Future Work

These are some topics which could be considered for future research projects:

- Additional equivalence-checking algorithms could be described and compared to the ones already contained in this report. For instance, it is possible that an algorithm to check equivalence of NFAs directly, without conversion to DFA format, may exist. Certainly equivalence algorithms for other types of finite automata could be considered, if models beyond DFAs and NFAs were to be researched. For large automata, approximate equivalence may even be of interest. As described above, it also may be possible to write a new algorithm using the inequivalence of any reachable states in the DFAs, not just the (in)equivalence of the start states.

- State complexity considerations could be discussed. For example, in the subset construction algorithm, it would be interesting to compare the size (number of states) of the original NFA against the size of the DFA which is typically produced. Different types of regular languages could be investigated to find best and worst case scenarios. These could be experimentally tested to empirically confirm the expected behavior.

- More work could be done on the JFLAP GUI to set up exams to be stored and executed within the program.
- JFLAP could also be extended to provide witnesses for regular expressions.
- It might be useful to set up encrypted JFLAP .jff answer files for students to test against their homework solutions. This would allow them to see a witness string as feedback, without revealing the full answer to the assignment. For now, "try" can meet this need.
- Significant performance improvements could be implemented in JFLAP. However, student NFAs are usually pretty small, so it may not make a significant difference to the user-perceived run time.
- Different equivalence-checking algorithm implementations could be coded and timed. It would be interesting to compare them on different categories of DFAs/NFAs. (Again, they would have to be fairly large models to show a significant impact on the run time.)
- The "try" version for Windows could be investigated, and setup instructions could be provided for the new grading tool.

References

[1] Marco Almeida, Nelma Moreira, and Rogério Reis. On the performance of automata minimization algorithms. Technical Report DCC-2007-03, Universidade do Porto, June 2007.

[2] Norbert Blum. An O(n log n) implementation of the standard method for minimizing n-state finite automata. *Information Processing Letters*, 57:65–69, 1996.

[3] Jean Bovet. Visual Automata Simulator - 1.2.1. Web site `http://www.versiontracker.com/dyn/moreinfo/win/35508`, 2006. Also at `http://www.cs.usfca.edu/~jbovet/vas.html`.

[4] Sang Cho and Dung T. Huynh. The parallel complexity of finite-state automata problems. *Inf. Comput.*, 97(1):1–22, 1992.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. McGraw-Hill, Boston, second edition, 2001.

[6] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R.E. Tarjan. Dynamic perfect hashing: upper and lower bounds. *Proc. 29th IEEE Annual Symposium on Foundations of Computer Science*, pages 524–531, October 1988.

[7] Finite state automaton applet. Montana State University Computer Science Department Web site. `http://www.cs.montana.edu/webworks/projects/fsa-old/fsa.html`.

[8] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, 1979.

[9] David Gries. Describing an algorithm by Hopcroft. Technical Report TR 72-151, Cornell University, December 1972.

[10] HashMap Java 2 platform standard edition 5.0 API specification. Sun Web site. `java.sun.com/j2se/1.5.0/docs/api/java/util/HashMap.html`.

[11] HashSet Java 2 platform standard edition 5.0 API specification. Sun Web site. `java.sun.com/j2se/1.5.0/docs/api/java/util/HashSet.html`.

[12] J. E. Hopcroft and R. M. Karp. A linear algorithm for testing equivalence of finite automata. Technical Report TR 71-114, Cornell University, December 1971.

[13] J. E. Hopcroft and J. D. Ullman. A linear list merging algorithm. Technical Report CU-CSD-71-111, Cornell University, November 1971.

[14] J. E. Hopcroft and J.D. Ullman. Set merging algorithms. *SIAM Journal on Computing*, 2(4):294–303, December 1973.

[15] John Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical Report STAN-CS-71-190, Stanford University, January 1971.

[16] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, Boston, second edition, 2001.

[17] JFLAP 6.4 tutorial. JFLAP Web site. `http://jflap.org/tutorial/`.

[18] Neil D. Jones, Y. Edmund Lien, and William T. Laaser. New problems complete for nondeterministic log space. *Mathematical Systems Theory*, 10:1–17, 1976.

[19] S. C. Kleene. Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, volume 34 of *Annals of Mathematics Studies*, pages 3–41. Princeton University Press, Princeton, NJ, 1956.

[20] Timo Knuutila. Re-describing an algorithm by Hopcroft. *Theoretical Computer Science*, 250:333–363, 2001.

[21] Ambrose Kofi Laing. BFLAP Perl source code. `http://www.cs.tufts.edu/~laing/bflap.html`.

[22] Hing Leung. CS 510 (Spring 2008) note #4. Class notes.

[23] Nelma Moreira and Rogério Reis. Interactive manipulation of regular objects with FAdo. *SIGCSE Bull.*, 37(3):335–339, 2005. The site at `www.ncc.up.pt/fado` is currently under construction.

[24] M.O. Rabin and D. Scott. Finite automata and their decision problems. *IBM J. Research and Development*, 3(2):115–125, 1959.

[25] B. Ravikumar and G. Eisman. Weak minimization of DFA: an algorithm and applications. *Theor. Comput. Sci.*, 328(1-2):113–133, 2004.

[26] Kenneth Reek. Educational software: Try. Rochester Institute of Technology Web site. `http://www.cs.rit.edu/~kar/software.html`.

[27] Matthew B. Robinson. The Java Computability Toolkit: A Java-based tool for models of computation. Master's thesis, SUNY Institute of Technology at Utica/Rome, July 1998. `http://humboldt.sunyit.edu/JCT/HTMLthesis/CONTENTS.htm`.

[28] Susan H. Rodger and Thomas W. Finley. *JFLAP: An Interactive Formal Languages and Automata Package*. Jones and Bartlett Publishers, Sudbury, Massachusetts, 2006.

[29] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, Boston, 1997.

[30] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time: Preliminary report. *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, pages 1–9, April 1973.

[31] Larry Joseph Stockmeyer. The complexity of decision problems in automata theory and logic. Ph.D Thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering, 1974. `http://hdl.handle.net/1721.1/15540`.

[32] Shunichi Toida. CS390 introduction to theoretical computer science. Old Dominion University Web site with interactive exercises. `http://www.cs.odu.edu/~toida/nerzic/390teched/web_course.html`.

[33] Gertjan van Noord. FSA6.2xx: Finite state automata utilities. Web site. `http://www.let.rug.nl/~vannoord/Fsa/`.

[34] Luiz Filipe M. Vieira, Marcos Augusto M. Vieira, and Newton J. Vieira. Language emulator, a helpful toolkit in the learning process of computer theory. *SIGCSE Bull.*, 36(1):135–139, 2004.

[35] Bruce W. Watson. A taxonomy of finite automata minimization algorithms. Technical Report 93/44, Eindhoven University of Technology, May 1994.

[36] Bruce W. Watson and Jan Daciuk. An efficient incremental DFA minimization algorithm. *Natural Language Engineering*, 9(1):49–64, 2003.

[37] Sheng Yu. Regular languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*. Springer, 1997. Available on his Web site at `http://www.csd.uwo.ca/~syu/public/draft.ps`.